UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA INFORMÁTICA

DESARROLLO DE UN CLASIFICADOR VISUAL DE ESPECIES DE AVES MEDIANTE REDES NEURONALES CONVOLUCIONALES

DEVELOPMENT OF A VISUAL CLASSIFIER OF BIRD SPECIES BY CONVOLUTIONAL NEURAL NETWORKS

Realizado por

Antonio Miguel Pérez Segarra

Tutorizado por

Ezequiel López Rubio

Karl-Khader Thurnhofer Hemsi

Departamento

Departamento de Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JULIO DE 2019

Fdo. El/la Secretario/a del Tribunal

Fecha defensa: de julio de 2019

Resumen

Se ha desarrollado un clasificador visual de especies de aves mediante redes neuronales convolucionales, en lenguaje *Python* haciendo uso de la librería *Keras*. Se dispone de un conjunto de datos de 5771 imágenes repartidas entre 67 clases distintas. Este conjunto de imágenes fue cedido por Antonio Miguel Pérez Ortigosa, a quien pertenece, para el desarrollo del presente trabajo. Las clases contempladas son especies de aves, diferenciando entre machos, hembras y juveniles en su etapa de comienzo de vuelo, allí donde sean diferenciables (p. e. dimorfismo sexual). La arquitectura de red usada es *InceptionV3*, la tercera versión de una sucesión de modelos de red diseñados por Google. Con el objetivo de agilizar aún más el aprendizaje y conseguir una mejor precisión, se hizo uso de un optimizador RMSProp y de una tasa de aprendizaje variable forzada cuando el entrenamiento se quede estancado.

El conjunto de datos inicial incluía imágenes con múltiples aves en pantalla o pocas imágenes por clase, por lo que se tuvo que proceder a una limpieza de datos. El conjunto de datos sufría de otro *handicap*, las imágenes no tenían la misma luminosidad unas que otras, incluso dentro de la misma clase. Se llevó a cabo la corrección de luminosidad de las imágenes mediante la normalización del histograma del canal L en el espacio de color CIELAB, pasándolas tras la normalización al espacio de color RGB otra vez.

Se ha implementado validación cruzada estratificada, con 10 folds, cumpliendo el objetivo de ser imparcial a la hora de generar un resultado. La precisión media del entrenamiento se queda en un 86.74 %.

Se desarrolló conjuntamente un interfaz gráfico de usuario con conexión a Wikipedia para extraer información de la especie clasificada.

Palabras clave

Red neuronal convolucional, ave, clasificación, Keras, Tensorflow, Kivy, visión por computador.

Abstract

A visual classifier of bird species by convolutional neural network has been developed, written in *Python* and using *Keras* library. There is a dataset compounded of 5771 images within 67 different classes. This dataset was given by Antonio Miguel Pérez Ortigosa, whom it belongs, in order to support the development of this project. The classes are bird species, distinguished between male, female and young specimens in their flight learning stage, there when the sexes can be differentiated (i.e.: sexual dimorphism). The used network arquitecture is *InceptionV3*, third version of a set of network models developed by Google. In order to speed the learning up and achieve a better accuracy, it has been used a RMSProp optimizer and an adaptative learning rate, forced to work when the training is stuck in a local minimum.

The initial dataset included multiple birds in some images or too few images in some classes, so it was necessary to perform a dataset cleaning. Furthermore, the dataset had another handicap, images were different one to another in luminosity terms, even between specimens of the same species. This promoted to normalize the images' luminosity by using histogram normalization in L channel in the color space CIELAB, parsing to RGB again before the normalization.

Stratified k-fold cross validation has been implemented, with $k = 10$, reaching the goal of being impartial at generating results. The average accuracy was 86.74 %.

Moreover, a graphic user interface was developed, with a direct Wikipedia connection in order to extract information about the classified species.

Keywords

Convolutional neural network, bird, classification, Keras, Tensorflow, Kivy, computer vision.

Índice general

Capítulo 1

Introducción

La clasificación de aves es un problema a la hora de aprender ornitología. En España, existen 347 especies de aves [1]. A la hora de aprender a reconocer estas aves por su morfología, puede resultar difícil dado el parecido sustancial entre algunas aves. Por ello, no solo se suele utilizar la vista cuando se quiere nombrar a un ave avistada. El canto difiere también entre ellas. Por poner un ejemplo, los páridos, como el carbonero común (*Parus major*, figura 1.1), el cual dispone de más de treinta notas en su repertorio de canto [2]. Este repertorio también depende de la zona de donde es el ejemplar, ya que se ve influenciado por matices territoriales.

Figura 1.1: Carbonero común (*Parus major*)

Además de esas 347 especies que existen, cada una de ellas puede tener dimorfismo sexual (diferencias morfológicas entre el macho y la hembra dentro de una misma especie), como por ejemplo la curruca capirotada (*Sylvia atricapilla*, figura 1.2), e incluso diferencias entre el ejemplar adulto y el joven volantón, como el petirrojo (*Erithacus rubecula*, figuras 1.3 y 1.4)

Figura 1.2: Curruca capirotada (*Sylvia atricapilla*; machos izqda., hembras dcha.)

Figura 1.3: Petirrojo volantón (*Erithacus rubecula*)

Figura 1.4: Petirrojo adulto (*Erithacus rubecula*)

Por tanto, además de ser difíciles de identificar mediante la morfología, también lo son por el canto. En este proyecto solo se aborda el reconocimiento de especies mediante la morfología. Para ello se aplican redes neuronales convolucionales, así como técnicas de visión por computador.

1.1. Motivación

La idea principal de este proyecto es tener una ayuda técnica a la hora de determinar aves observadas en una jornada. A la hora de buscar una especie que no es conocida por el observador en una guía de aves, se debe orientar por las descripciones que ofrezcan en la misma, además de utilizar los dibujos como referencia. Si un programa de ordena-

dor pudiera catalogarla directamente o, al menos dar un nombre de una especie similar morfológicamente, sería una tarea menos ardua; sobre todo después de un día de muchas observaciones a la hora de hacer un listado de las aves vistas, costumbre que siguen la mayoría de ornitólogos. Otro punto interesante es que se podría utilizar esta plataforma, mediante un sistema colocado en algún lugar de interés, para censar las aves de esa zona, e incluso aligerar la carga de hacer un informe de impacto ambiental en caso de que se quisiera edificar o construir en cierto sitio.

1.2. Objetivos

El objetivo general de este proyecto es desarrollar, mediante redes neuronales convolucionales, un clasificador automático de aves a partir de fotos de individuos. Más concretamente:

- Desarrollar un clasificador preciso, que sea capaz de discernir con facilidad entre las 67 clases que se abordan en este proyecto.
- Utilizar técnicas de visión por computador para que todas las imágenes estén normalizadas, y con eso,
- Evitar problemas que tengan origen en la luminosidad de las imágenes.
- Desarrollar un interfaz gráfico de usuario lo más amigable posible, para que con ello sea fácil el proceso de predicción mediante la red entrenada.

1.3. Estado de la técnica

Las redes neuronales convolucionales se llevan usando para clasificación de imágenes desde el año 2012. Uno de los conjuntos de datos más usados para competiciones, por ejemplo, es el de ***ImageNet***. Consiste de 15 millones de imágenes repartidas entre 22000 clases diferentes, aunque en el concurso ***ImageNet LSVRC-2010*** [3] sólo se tomaron 1.2 millones de imágenes en 1000 clases.

Existen numerosas investigaciones relacionadas con el poder reconocer animales en imágenes. Un ejemplo sería la implementación y comparación de las redes neuronales convolucionales con respecto a otros métodos de reconocimiento de imágenes que hicieron

en la Universidad de Zilina, Eslovaquia [4]. En este caso tomaron un conjunto de datos compuesto de 500 imágenes en 5 clases distintas.

En el estado del arte ya existen aplicaciones que reconozcan aves, como por ejemplo **Google Lens**[5] o **Merlin Bird ID**[6]. Sin embargo, el primero no ofrece toda la información necesaria para identificar un ave (suele reconocer sólo el género taxonómico, p. e. decir que un Papamoscas cerrojillo [*Ficedula hypoleuca*] es un papamoscas, simplemente); y el segundo es bastante completo, ya que se nutre de una Base de Datos que van rellenando los usuarios de **eBird**[7] (plataforma online que sirve de "libreta de campo" digital), pero, por contra, sólo funciona en dispositivos móviles (*iOS/Android*).

Un grupo de estudiantes de la Universidad de Richmond [8] desarrollaron un clasificador visual de aves de Estados Unidos, y junto al proyecto desarrollaron un comedero con una cámara apuntando hacia el mismo, con el objetivo de, cuando se detecte movimiento, fotografiar al ave o las aves que se encuentren en escena y alimentar su conjunto de datos. Para la clasificación de aves usan ***Faster Region-based Convolutional neural network***, que extrae regiones de la imagen significativas y las procesa con una red neuronal convolucional, lo que le permite identificar varias aves en una misma foto, aunque sean distintas. Además, este proyecto utiliza, aparte de la morfología como tal de las aves, sus diferencias de tamaño (ya que las fotos están tomadas todas a la misma distancia de las aves, lo que respeta la proporción de cada una). En el caso del proyecto que hemos desarrollado, no contamos con esa "ayuda" a la hora de clasificar.

En España, la **Sociedad Española de Ornitología (SEO)**[9] desarrolló una aplicación de ayuda al reconocimiento de aves mediante preguntas, al estilo de árbol de decisión. Sin embargo, este proyecto cubre el reconocimiento de la figura del ave y su clasificación por la misma. Además, tiene en cuenta el dimorfismo sexual y la madurez del individuo.

Capítulo 2

Redes Neuronales Convolucionales y Visión por Computador

Las redes neuronales convolucionales [10], más conocidas por sus siglas en inglés **CNN (Convolutional Neural Network)**, son un tipo especializado de red neuronal artificial para el procesamiento de información que tenga una topología en rejilla o *"grid"*. En el caso que concierne a este proyecto, la información viene dada en imágenes. Las imágenes (RGB) no son sino matrices tridimensionales que en cada una de sus "celdas" almacenan un valor comprendido entre 0 y 255. Cada una de las dimensiones de esa matriz corresponde con uno de los canales de color del espacio de color en el que se encuentre la imagen. En las imágenes RGB, los canales son rojo, verde y azul (del inglés **R**ed, **G**reen y **B**lue).

2.1. Capas de convolución (*CNN*)

Una CNN se compone de un gran número de capas (de ahí el apelativo de **Deep Learning**). La red se suele distribuir en bloques de dos capas: **Capa de convolución y Capa de *pooling***. Las capas de convolución se encargarán de extraer las características que serán analizadas por las capas totalmente conectadas o *fully-connected*

2.1.1. Capa de convolución

La capa de convolución se subdivide en dos etapas: **etapa de convolución y etapa de activación**.

La convolución, en términos generales, es una operación entre dos funciones con un número

real por argumento, representada como $(i*k)(t) = o(t)$. En el caso de las redes neuronales, la primera función (en el caso anterior $i(t)$) es la **entrada (*input*)**, así como la segunda (en este caso $k(t)$) es el ***kernel***. A la salida se le denomina **mapa de características (*feature map*)**. La entrada suele ser un vector multidimensional de datos; y el kernel, un vector multidimensional de parámetros que van siendo modificados por el algoritmo de aprendizaje. Este aprendizaje no se va haciendo de golpe, sino que se van modificando los pesos por una tasa de aprendizaje (*learning rate*). Cuanto mayor es esta tasa, más rápido aprende la red porque se acerca con más facilidad a los pesos óptimos, pero tiene el riesgo de quedarse entre dos pesos atorada; en cambio el uso de una tasa de aprendizaje pequeña tiene más probabilidades de converger, pero puede tardar demasiado en encontrarlos. A los vectores multidimensionales se les conoce como **tensores**. El kernel es lo que se relacionaría con los pesos sinápticos en el caso de la red neuronal más simple, el *perceptrón* (figura 2.1).

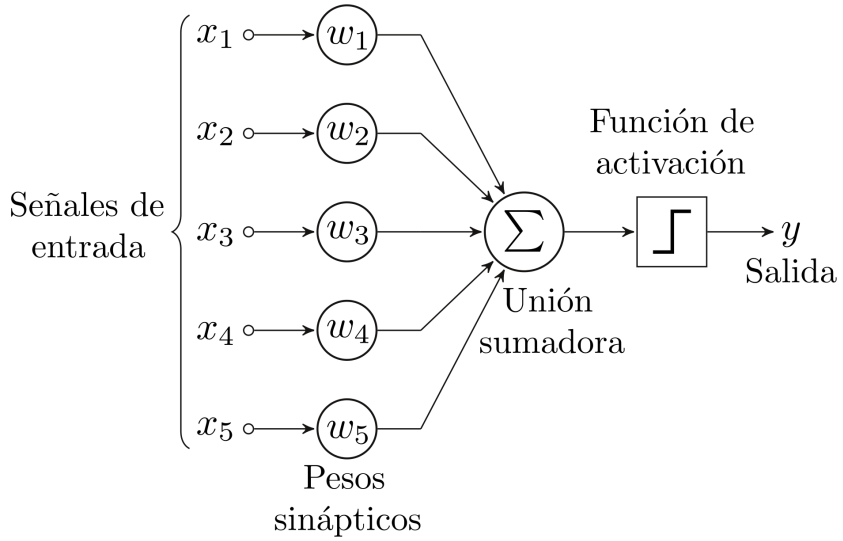


Figura 2.1: Perceptrón simple con 5 señales de entrada y función *step* como función de activación [11].

En resumen, en esta etapa se realizan varias convoluciones en paralelo para producir un conjunto de activaciones lineales.

La etapa de activación es la más simple, ya que su único propósito es pasar las activaciones lineales de la etapa anterior por una función de activación no lineal, como por ejemplo la función sigmoide o la función **ReLU**. Hace unos años la más usada era la función sigmoide (figura 2.2), pero con el surgimiento del *Aprendizaje profundo* y su dificultad de

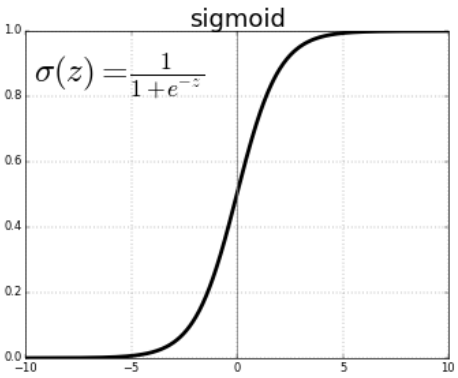


Figura 2.2: Función sigmoide [12]

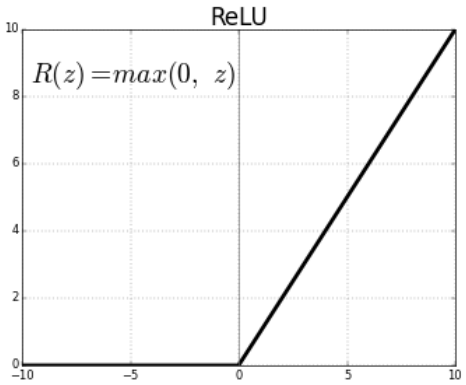


Figura 2.3: Función ReLU [12]

entrenamiento por el problema de desaparición de gradiente se empezó a usar la función ReLU [12].

El **problema de desaparición de gradiente** se encuentra en el entrenamiento de redes neuronales artificiales con métodos de entrenamiento basados en el gradiente del error y con **backpropagation** (método de propagación del error dirección salida-entrada, cambiando los pesos sinápticos de la red). Esto conlleva a que, cuantas más capas tiene una red neuronal, más se reducen los cambios en los pesos cuanto más cerca de la entrada de la red se encuentren, ya que los cambios son proporcionales a la derivada parcial de la función de error con respecto al peso en concreto que se esté modificando. Por ello, si el número de capas es suficiente, algunos pesos no se verán alterados, lo que entorpece en gran medida el entrenamiento [13].

Para subsanar el error, se creó la antes comentada función ReLU (*Rectified Linear Unit*, figura 2.3) que, gracias a su simpleza es la usada actualmente, si no una de sus variantes. Esta función permite el paso de todos los valores positivos tal cual sean, y asigna a todos los valores negativos el valor cero.

También se intenta agilizar el encontrar el mínimo error, es decir, agilizar la búsqueda de la convergencia. Con ese objetivo, se usan los algoritmos de optimización. Estos algoritmos de optimización son unas funciones matemáticas que dependen de los parámetros internos del modelo.

2.1.2. Capa de *Pooling*

En la capa de *pooling* se usa la función homónima para modificar la salida de la misma. Una función *pooling* reemplaza la salida de la red en determinados sitios con un resultado estadístico que resume las salidas cercanas. Por ejemplo, la operación de ***max pooling*** da como resultado la salida máxima dentro de una vecindad rectangular (ver ejemplo en figura 2.4).

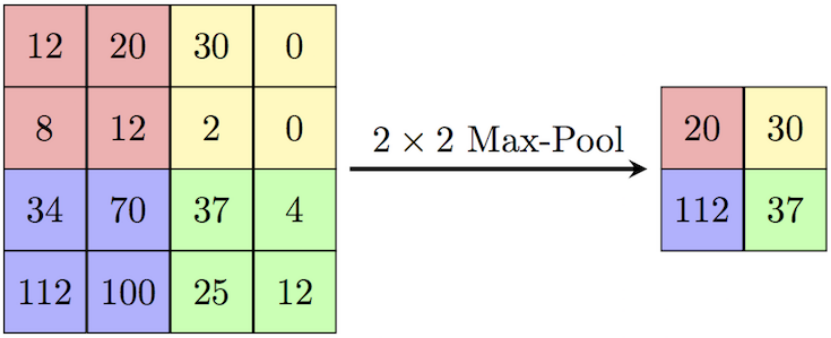


Figura 2.4: Ejemplo de la operación ***max pooling*** para cada cuatro salidas vecinas [14]

El *pooling* sirve para hacer la representación prácticamente invariable frente a pequeñas traslaciones en la entrada. Esto significa que si la entrada es movida un poco, los valores de la mayoría de las salidas tras la función *pooling* se mantienen igual. Es decir, es útil cuando se quiere saber si una determinada característica existe en la imagen, más que saber exactamente donde se encuentra esa característica. Por ejemplo, para determinar un ave, no se sabe dónde está el animal a nivel de coordenadas, pero si se sabe que si está, estará compuesto de un pico, alas, patas, cola, etc.

Además, con el *pooling*, cada vez que se aplica las entradas a la siguiente capa las dimensiones disminuyen una o varias unidades, por lo que resulta en una mejor eficiencia de la red y una reducción importante de memoria requerida para almacenar los parámetros.

2.2. Capas *flattening*, densas (*fully-connected*) y salida

Las capas *flattening* y densas son las que se encuentran justo antes de la capa de salida, y son las que más se asemejan a una red neuronal artificial sin el apellido de convolucional.

2.2.1. Capa *Flattening*

Una vez la entrada original ha pasado y ha sido modificada por todas las capas anteriores y se obtiene un mapa de características del último *pooling*, el próximo objetivo es transformar la matriz a una única columna con el objetivo de que sea la entrada de la red de capas totalmente conectadas (densas o *fully-connected*)[15] (figura 2.5).

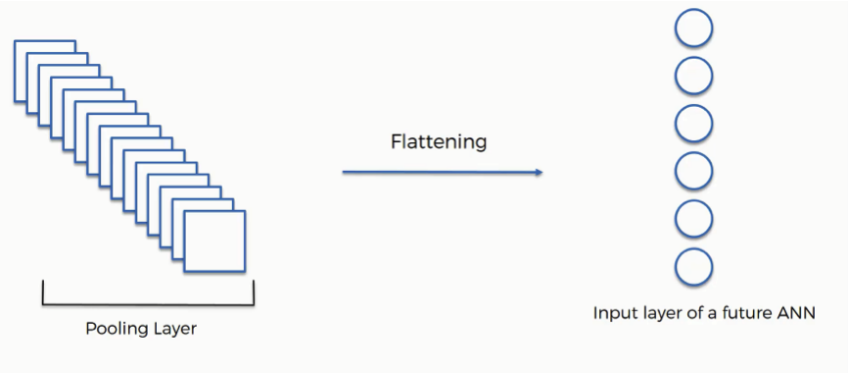


Figura 2.5: Ejemplo de aplanamiento de la última capa de pooling, para dar lugar a la entrada a la red de capas densas [16]

2.2.2. Capas densas o *fully-connected*

Estas capas consisten en que cada una de las neuronas de una capa está conectada con todas las neuronas de la siguiente (como ocurre en el perceptrón multicapa). En resumen, es una red neuronal artificial donde las salidas de la capa anterior son cada una de las entradas de la capa actual (representación gráfica de la relación entre capas densas en la figura 2.6).

2.2.3. Capa de salida

Cuando se obtiene un resultado de estas últimas capas, se tiene un vector de números reales que se pasarán por una función ***Softmax*** (o *normalized exponential function*, figura 2.7). Esta función toma la entrada del vector de N números reales (donde N es el número de clases que se quiere que la red clasifique) y lo normaliza en una distribución de probabilidad que consiste en N probabilidades, que será la predicción que haya realizado la red. El máximo porcentaje será para la clase a la que es más probable que pertenezca la entrada inicial. Por tanto, la representación gráfica de la red completa podría ser como

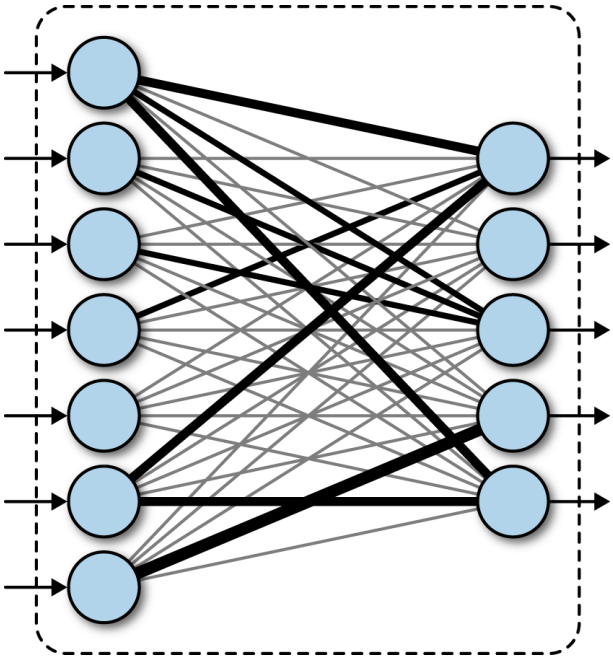


Figura 2.6: Representación de una capa densa en una red neuronal artificial [17]

$$\sigma : \mathbb{R}^K \to [0,1]^K$$
$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{para } j = 1, \ldots, K.$$

Figura 2.7: Función *Softmax* [18]

la figura 2.8.

2.3. Modelos de red

Una vez que se ha descrito como funcionan las capas que componen las CNNs, lo siguiente es saber cómo distribuir estas capas, que reducciones de tamaños de las matrices o cambios de número de parámetros se van a hacer, etc.

Dentro de las arquitecturas de las CNN, cada una tiene su propia eficiencia y fiabilidad, que son los dos valores a comparar entre ellas.Como se puede observar en la figura 2.9, aparece una comparativa de las 14 arquitecturas de red más utilizadas en la actualidad. En el eje de abcisas están representadas las operaciones que realiza la red en un sólo paso (lo que da una idea de cuán rápida es comparada con las demás en una misma máquina); mientras que en el eje de ordenadas se tiene la precisión de estas redes a la hora de predecir. Por tanto, se podría decir de manera objetiva que cuanta más precisión

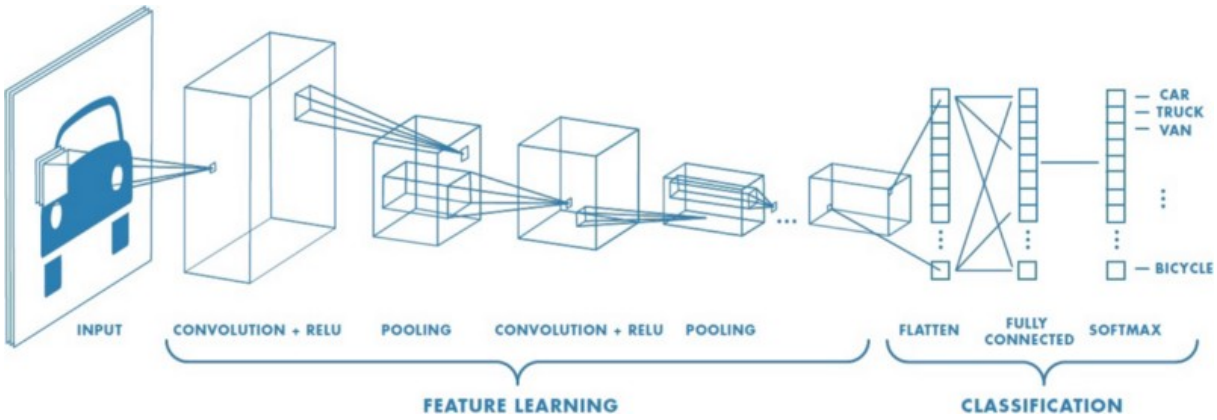


Figura 2.8: Ejemplo de arquitectura de red [19]

disponga la arquitectura respecto a un menor número de operaciones, mejor será respecto a las demás. Muchas de las que salen en el gráfico son re-estructuraciones y mejoras de la misma. Como ejemplo, *GoogLeNet* sería la *Inception-V1*, que con respecto a su versión 3, mejora sustancialmente su precisión a coste de aproximadamente triplicar el número de operaciones.

2.4. Cambios en las características de las entradas

En el caso de las CNN estas entradas como se ha descrito anteriormente pueden ser imágenes. En las imágenes, cuando existen cambios de luminosidad o contraste, por ejemplo, cambian mucho los valores de la matriz y, por tanto, se diferencian bastante unas imágenes de otras aunque sean a un mismo objeto. Por ello, se suelen aplicar métodos de visión por computador que hagan más fácil el identificar un determinado sujeto en una misma clase a la que pertenece.

Uno de los parámetros a normalizar suele ser la luminosidad. A la hora de detectar colores en una imagen oscura, esta suele tener los colores más apagados que una foto del mismo sujeto en mejores condiciones de luz. Por tanto, si se normalizara la luminosidad de todas las fotos de un dataset, estas tendrían colores más similares que de no tratarse.

Empezando con la normalización de luminosidad, una de las técnicas más utilizadas es la ecualización del histograma. Las imágenes en blanco y negro solo tienen una dimensión de color, es decir, en términos de computación sería una matriz bidimensional, donde cada uno de los valores es la intensidad del píxel; cuanto más cercano a 0, más oscuro, y conforme el valor se acerca a 255 (su valor máximo), más claro. Por tanto, la técnica

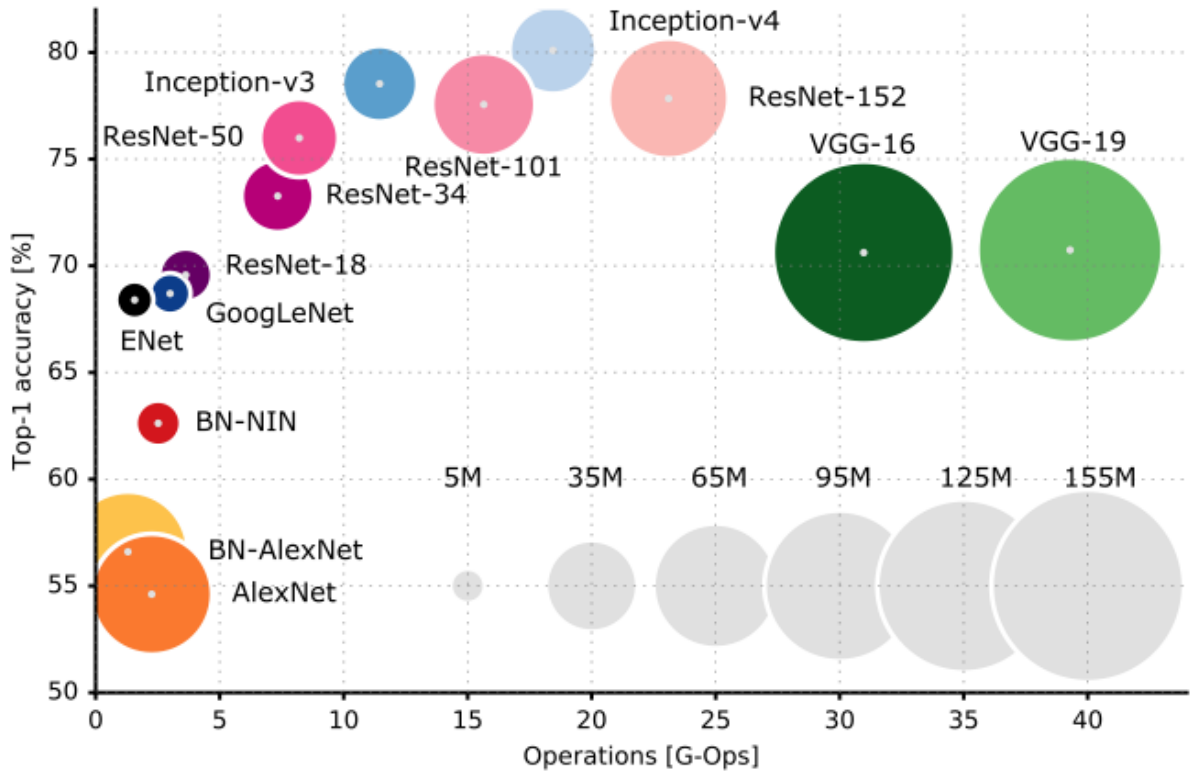


Figura 2.9: Comparativa entre arquitecturas de CNN [20]

de ecualización del histograma consiste en dispersar la intensidad del contraste de los píxeles de una imagen, para que aquellas zonas que tengan poco contraste le aumente. Con esto se consiguen imágenes en las que se distinguen mejor los objetos que contienen. A modo de ejemplo, la figura 2.10 representa una imagen sin tratar y una tratada, así como sus histogramas. También han surgido modificaciones de este algoritmo que se encargan de varios sub-histogramas a la vez, como por ejemplo el método **CLAHE (*Contrast Limiting Adaptive Histogram Equalization*)**. Con esto se consigue que solo se vean modificadas las zonas con "peor" contraste de la imagen, y no toda en conjunto.

Una técnica de normalización de luminosidad en imágenes a color (p.ej.: RGB) se lleva a cabo cambiando al espacio de color ***CIELAB*** (también conocido como ***CIE L*a*b****; **L** - *luminosidad*, **a** - *desde el verde al rojo* y **b** - *desde el azul al amarillo*, representado gráficamente en la figura 2.11) y aplicando la normalización de histograma al canal que trata la luminosidad.

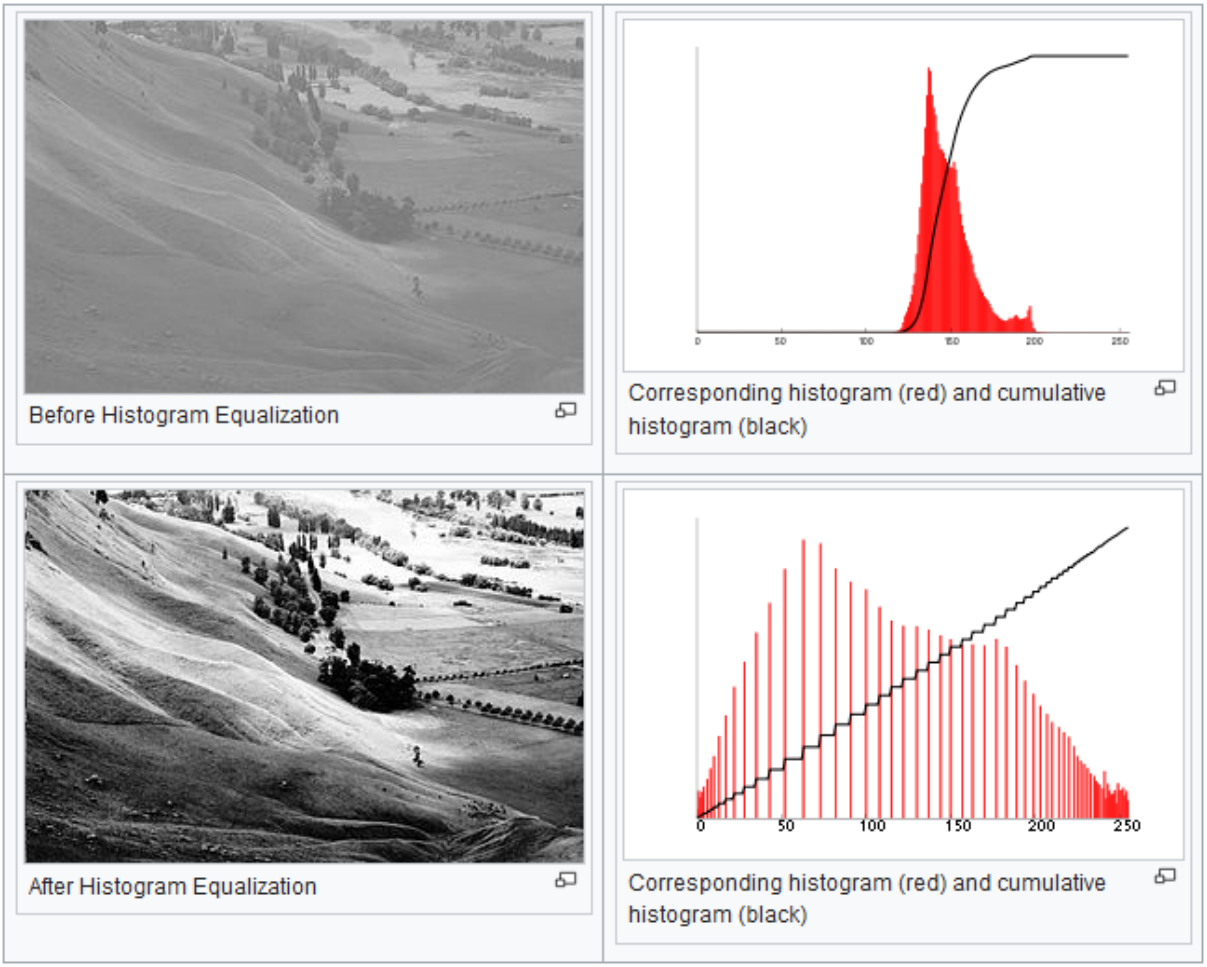


Figura 2.10: Misma imagen, sin tratar arriba, tratada abajo; y sus respectivos histogramas [21]

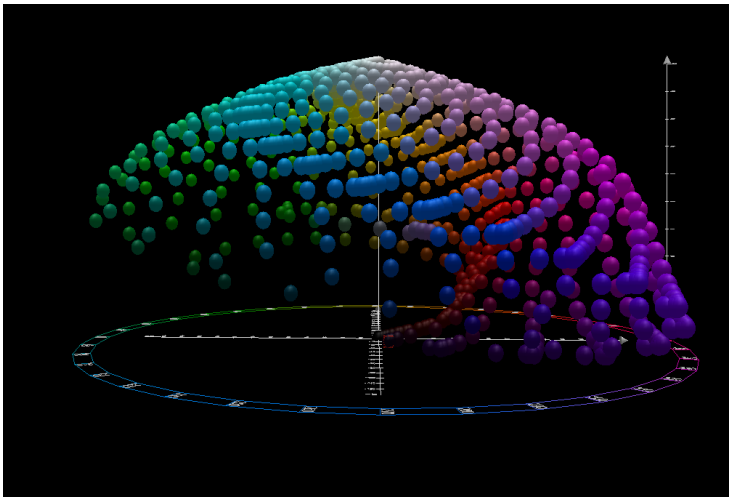


Figura 2.11: Espacio de color CIELAB representado en 3D con una vista frontal del mismo [22]

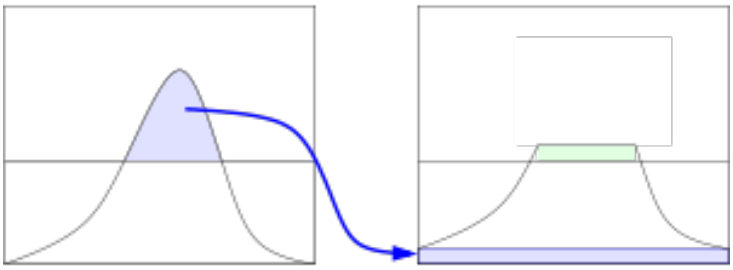


Figura 2.12: Ejemplo de funcionamiento del algoritmo CLAHE cuando se supera el límite de contraste [23]

2.4.1. CLAHE

El método CLAHE (*Contrast Limited Adaptive Histogram Equalization*) es una mejora del método AHE (*Adaptive Histogram Equalization*). AHE consiste en sacar varias regiones de la imagen y normalizar los histogramas de esas regiones por separado, lo que hace que aumenten más de contraste las regiones más oscuras de la imagen y viceversa. CLAHE surgió para solucionar el problema de AHE, amplificar demasiado regiones cuyo contraste es casi constante, ya que el histograma en estas zonas suele estar muy concentrado. CLAHE pone un límite en el histograma y si se supera, el valor superado es distribuido de manera uniforme por todo el histograma (figura 2.12) [23].

2.5. Modelo-Vista-Controlador

En este proyecto se desarrolla una interfaz gráfica. Por ello, es necesario establecer una técnica de modelado de la arquitectura software de las que existen, como descomposición modular, modelo-vista-controlador (que será la abordada aquí), etc.

El estilo de arquitectura software ***Modelo-Vista-Controlador*** **(MVC)** se define como aquel que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos [24].

Las funciones como tal de la aplicación, que no se relacionan con el apartado visual o estético, son el modelo. Es el que se encarga de controlar los datos que son propios del sistema. El modelo solo se relaciona y comunica directamente con el controlador. El controlador se encarga de traducir los eventos y las entradas desde la interfaz al modelo, y este, a su vez cuando termine el procesado de lo que le pase, devolverá un resultado que podrá ser transmitido, o no, a la interfaz. La interfaz o vista es la fachada que se

puede observar en pantalla de la aplicación. Esta última se comunica directamente con el controlador, para desencadenar eventos; o con el modelo, para conseguir la información que dará como salida (representación del patrón en figura 2.13).

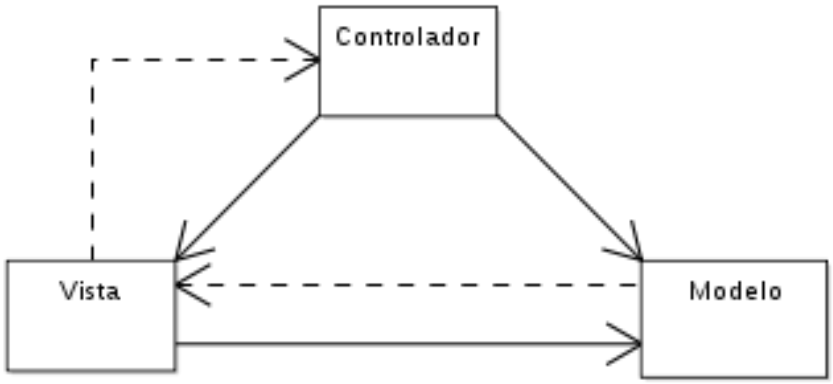


Figura 2.13: Representación gráfica sencilla del MVC. Líneas continuas: comunicación directa; líneas discontinuas: comunicación indirecta [25]

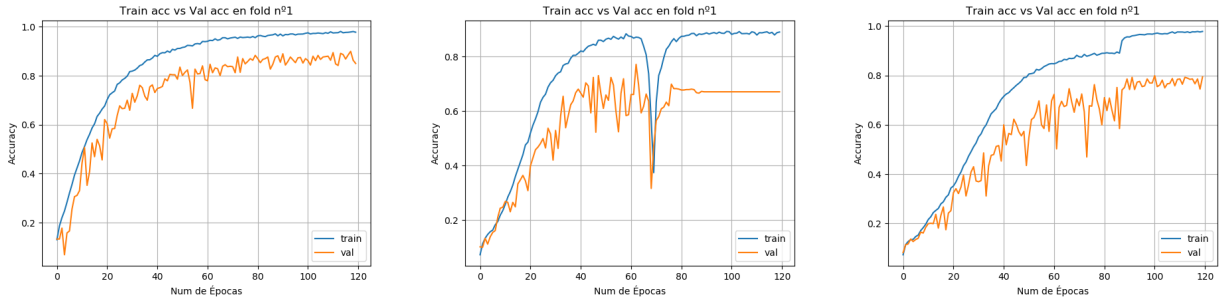
Capítulo 3

Diseño del sistema y pruebas

A la hora de diseñar el sistema clasificador, hay que tener en cuenta factores ya narrados previamente y hacer pruebas con el objetivo de asegurarse de que va a funcionar en las mejores condiciones que se le puedan dar. Para ello hay que elegir una arquitectura de red acorde para no quedarse corto con la precisión en casos de gran cantidad de clases como en este proyecto. También se tiene en cuenta que no todos los posibles usuarios de la aplicación final deben tener conocimientos de ejecución de una red neuronal de estas características para su uso en predicción, por lo que se desarrolló una interfaz de usuario para hacer este paso más amigable para el posible futuro usuario.

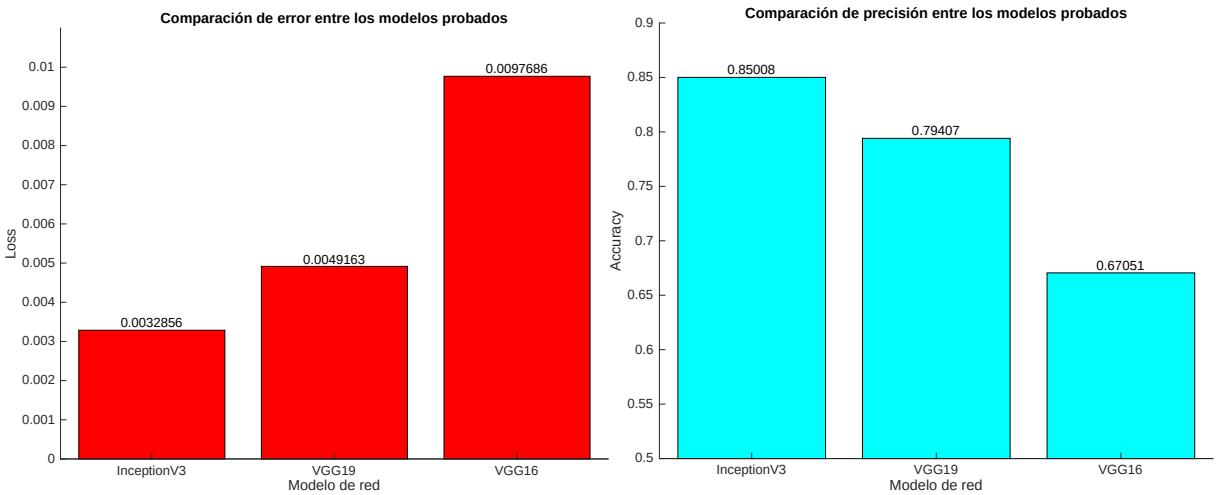
3.1. Arquitectura de red

Como se explicó en secciones anteriores, una de las decisiones a realizar a la hora de implementar una red neuronal es cuál va a ser la arquitectura a usar. Con las primeras pruebas se implementa la arquitectura *VGG16* y su modificación *VGG19*. Estas arquitecturas no son las mejores, pero eran las más fáciles a la hora de implementar. Tras las pruebas con estas arquitecturas, se descartan por su inestabilidad y poca precisión. Por ello, se procedió a investigar sobre arquitecturas más eficientes. Se eligió InceptionV3 por su relación media de precisión/eficiencia. Las figuras 3.1a, 3.1b y 3.1c representan los entrenamientos de 120 épocas con los mismos parámetros (menos el tamaño de entrada) de InceptionV3, VGG16 y VGG19, respectivamente. También se representó gráficamente las diferencias de precisión y error entre los 3 modelos probados, observable en las figuras 3.2b y 3.2a.



(a) Gráfica resultado de InceptionV3 (b) Gráfica resultado de VGG16 (c) Gráfica resultado de VGG19

Figura 3.1: Resultados de los entrenamientos con distintos modelos de red



(a) Comparativa error (b) Comparativa precisión

Figura 3.2: Resultados de las métricas en los entrenamientos de los modelos

Respecto a precisión en entrenamiento prácticamente los tres consiguen la misma precisión. Sin embargo, cuando observamos precisión de validación, que es calculada con un conjunto de datos previamente apartado, se puede confirmar la superioridad de InceptionV3. Además, en los dos modelos VGG se puede ver como el aprendizaje es mucho más errático, con descensos en el caso de VGG16 que sólo podrían ser índice de sobre-ajuste cuando realmente después de unas épocas parece que recupera. En VGG19 se intuye como se hace un cambio obligado de tasa de aprendizaje en torno a la época 83 u 84, ya que cuando parecía que el entrenamiento se iba a estancar, da un salto en precisión, y la gráfica de validación se consigue estabilizar con menos alteraciones. Por todo ello, se considera que InceptionV3 es un modelo que se adapta mejor a el conjunto de datos que se trata en este proyecto y a las condiciones del mismo, por lo que se elige para el entrenamiento final.

3.1.1. Inception V3

Esta arquitectura, creada por Google, es la tercera versión en la serie de arquitecturas de redes neuronales convolucionales de aprendizaje profundo de Google. Su estructura está representada en la figura 3.3.

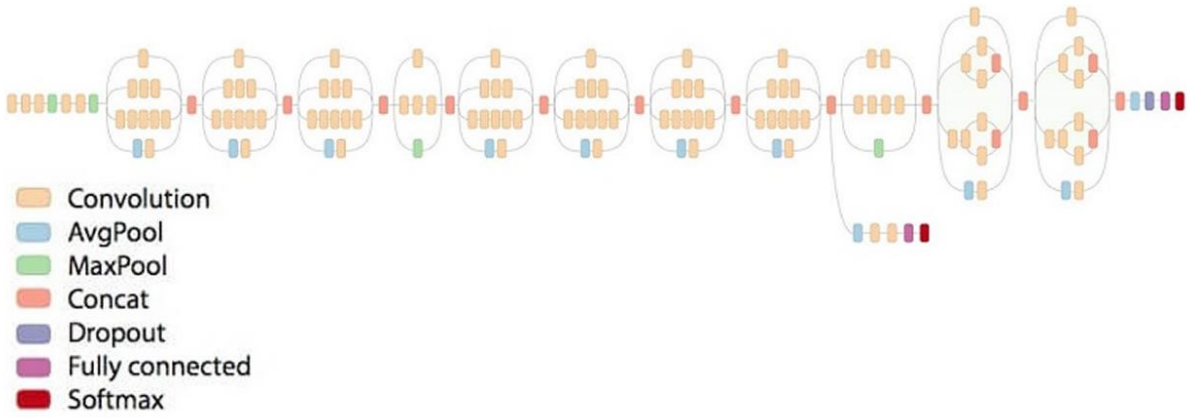


Figura 3.3: Arquitectura Inception V3 [26]

Como puede observarse en la figura 3.3, Inception V3 tiene dos "salidas" *Softmax*. La que está en la parte superior en la representación es la salida de la rama principal, que sería la salida real de la red. Por otra parte, la salida representada en la parte inferior de la figura corresponde a una rama auxiliar, denominada **clasificador auxiliar**. Este clasificador auxiliar está diseñado para empujar los gradientes más útiles hacia las capas

más tempranas de la red para hacer que sean útiles para el aprendizaje desde un principio, y no haya que esperar a que el algoritmo de aprendizaje las haya modificado lo suficiente como para que lo sean si fuera una única rama. Con esto se combate el **problema de desaparición de gradiente**, se promueve que el aprendizaje sea más estable y una mejor convergencia[27].

La implementación de esta arquitectura usando **Keras** es de la siguiente forma:

```
model = InceptionV3(include_top=True, weights=None,
                                input_tensor=None, classes=num_classes)
```

Los argumentos son los siguientes:

- **include_top**: Booleano que sirve para determinar si la red debe llevar la capa *fully-connected* o no. En este caso sí se necesita, ya que vamos a usar la arquitectura tal cual está diseñada.
- **weights**: Keras contiene pesos pre-entrenados para las arquitecturas. En el caso de InceptionV3 los pesos corresponden al dataset ImageNet (un millón de imágenes repartidas entre mil una clases). Este proyecto incluye el entrenamiento desde cero de la red, por lo que no se toma en cuenta el pre-entreno.
- **input_tensor**: Se utiliza para darle un tamaño de entrada específico a la red. Se usa el por defecto ($299 \times 299 \times 3$, imágenes de 299×299 píxeles y 3 canales de color).
- **classes**: Número de clases para el que se va a entrenar la red. Este proyecto ocupa la clasificación de **67** clases (con 5771 imágenes repartidas entre ellas en el dataset).

Junto al uso de la arquitectura *Inception V3* para el modelo, se acompaña el uso de un optimizador, concretamente ***RMSprop***.

3.1.2. RMSprop

Esta función fue propuesta por *Geoff Hinton* [28]. La tasa de aprendizaje, si se mantuviera en el mismo valor, el aprendizaje siempre avanzará con la misma velocidad y convergerá o no, dependiendo del conjunto de datos con el que se está tratando, además de intervenir el azar. Sin embargo, con el uso de la **normalización de mini-paquetes (*mini-batch normalization*)**, nombre que recibe la propia función *RMSprop*, la tasa

de aprendizaje es dividida por la media de los gradientes de error al cuadrado, con lo que va decayendo conforme nos acercamos a la solución. Este algoritmo "desciende" del anterior **Rprop**, el cual tenía problemas a la hora de funcionar con datasets grandes. La implementación con Keras quedaría como en la figura 3.4.

```
opt = RMSprop(lr=0.00005)
```

Figura 3.4: Definición del optimizador RMSProp en Keras

Donde `lr` es la tasa de aprendizaje (*learning rate*). El valor tan pequeño fue decidido con prueba y error, hasta que se logró que el algoritmo de aprendizaje convergiera en un número aceptable de épocas.

3.1.3. Tasa de aprendizaje variable

Con el objetivo de que el entrenamiento de la red no quede atrapado en un mínimo local, se especificó una tasa de aprendizaje variable (figura 3.5). Con esto se consigue que conforme se va quedando trabada la red en determinados errores, ella misma sea la que reduzca su tasa de aprendizaje para conseguir sortearlos. Se ha especificado que, si en la ejecución se encuentra con que se obtiene el mismo error 10 veces, se procede a reducir la tasa de aprendizaje a un 10 % de su valor en el momento. De esta forma se aprende con velocidad al principio para acercarse a su mínimo global, sin embargo cuando se aproxima puede seguir acercándose sin temor a que caiga en mínimo local.

```
reduce_lr_loss = ReduceLROnPlateau(monitor='loss', factor=0.1,
            patience=patience_lr, verbose=1, min_delta=1e-6, mode='min')
```

Figura 3.5: Función decreciente para la tasa de aprendizaje

3.2. Pre-procesamiento de la entrada

El conjunto de datos fue capturado mediante dos cámaras distintas que diferían en resolución:

- Canon EOS 400D, 10.1 MP
- Canon EOS 80D, 24.2 MP

Las imágenes del conjunto de datos usadas en este proyecto han sido tomadas indistintamente del momento del día o el tiempo meteorológico que hiciera. Por tanto, algunas fotografías son diferentes de otras que representen sujetos de la misma especie. Se puede observar lo comentado en las figuras 3.6 y 3.7.

Figura 3.6: Ejemplar de *Phoenicurus phoenicurus* en condiciones de alta luminosidad

Figura 3.7: Ejemplar de *Phoenicurus phoenicurus* en condiciones de baja luminosidad

Para subsanar este problema, se optó por usar un algoritmo de normalización de histograma en el espacio de color CIELAB. El algoritmo basa su funcionamiento en el canal L del espacio mencionado. Mediante el uso de *OpenCV* se pueden cambiar de espacio de color las imágenes para su posterior normalización del canal (código en 3.8). Una vez cambiado, simplemente hay que aplicar una normalización de histograma al canal L como si fuera una foto en blanco y negro cualquiera; y tras esto, pasarla otra vez a RGB que es el espacio con el que trabaja la red.

```
lab[:,:,0] = cv2.normalize(lab[:,:,0], None, alpha=0, beta=100,
                           norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
```

Figura 3.8: Normalización del canal L

Poniendo como ejemplo otra vez las figuras 3.6 y 3.7, si se ejecuta el algoritmo se pueden observar los resultados en las figuras 3.9 y 3.10.

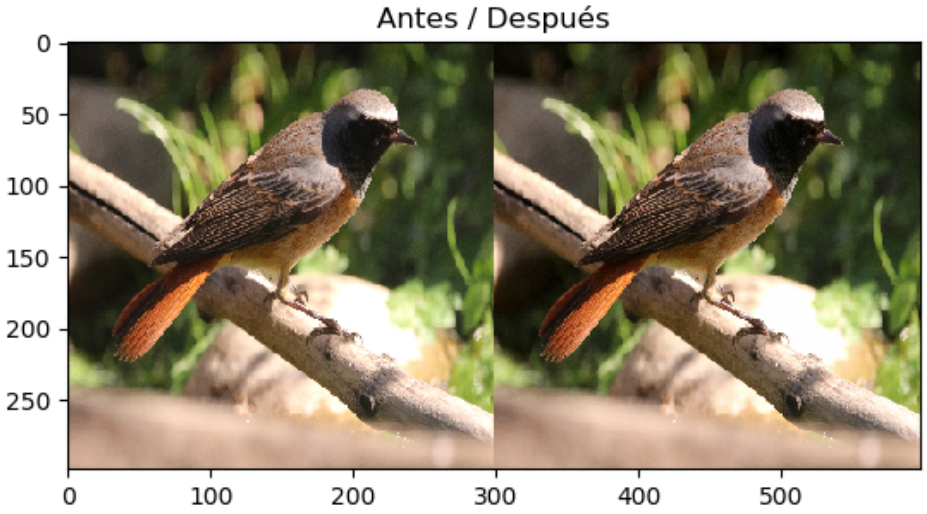


Figura 3.9: Comparativa entre la imagen clara sin tratar y tratada

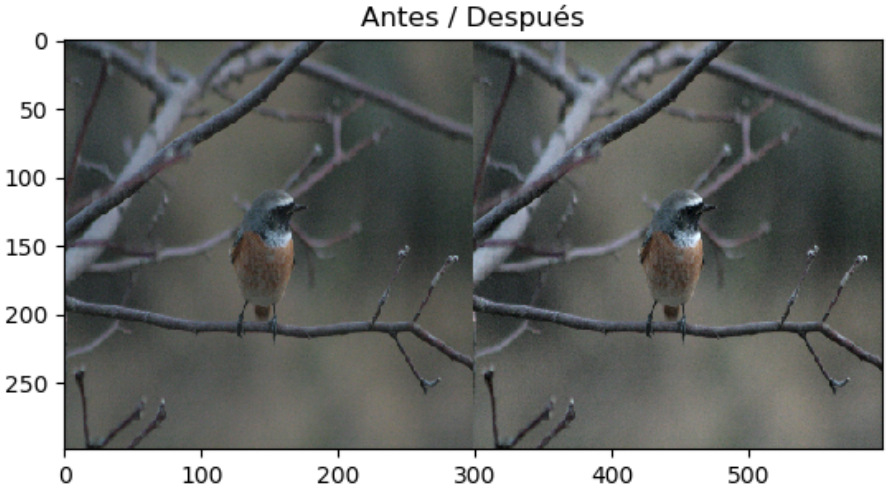


Figura 3.10: Comparativa entre la imagen oscura sin tratar y tratada

A simple vista solo se podría observar cierta mejoría en la foto oscura, donde los colores se ven más nítidos. En el caso de la foto con alta luminosidad no se aprecia más que un ligero retoque a los tonos oscuros de la imagen. Con este tratamiento se intenta normalizar los colores de las imágenes para que los colores de los especímenes solo dependan del sujeto en sí y no de la luz ambiental, en la medida de lo posible.

El efecto de la normalización puede ser mejor apreciado en la representación de los histogramas de color.

Teniendo en cuenta lo que ve el ojo humano, a la hora de compararlo con los resultados reales en el histograma, se intuye que la imagen clara ha sido más modificada que la oscura. Los niveles de color han sido rebajados en mayor medida que aumentados en la imagen

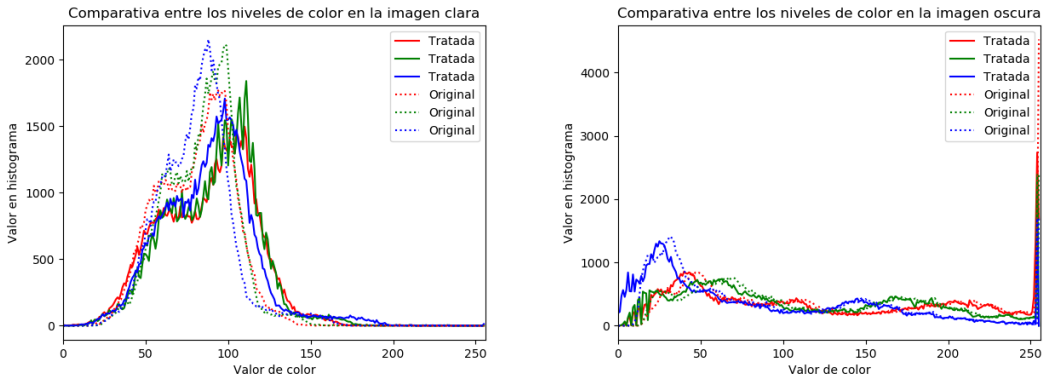


Figura 3.11: Histogramas de color de las imágenes, tratadas y originales

oscura, como bien se puede observar en la figura 3.11.

3.3. Validación cruzada estratificada (*Stratified K-fold cross-validation*)

Para garantizar que los resultados que se obtengan del entrenamiento sean independientes del conjunto de entrenamiento y del de validación que se escoja, se ha implementado la técnica de validación cruzada estratificada. La estratificada difiere de la que no tiene este apelativo en que la selección de datos la hace de manera que todas las clases tengan la misma proporción de imágenes, por lo que el resultado no se dejará llevar tanto por descompensación de imágenes entre clases. Con el objetivo de que el proyecto sea lo más justo posible a la hora de establecer un porcentaje de acierto, se ejecutará un 10-fold y posteriormente se calculará una media de las características de cada entrenamiento.

El entrenamiento, con la validación cruzada descrita, se desarrolló en el equipo siguiente:

- CPU: Intel Core i5-7400, 3GHz
- RAM: 8 Gb
- GPU: NVidia GeForce GTX 1060 6GB
- SO: Ubuntu 18.04

Para mayor comprensión de la técnica, observar la figura 3.12, donde viene representada gráficamente aquélla con $k = 4$ (4-fold).

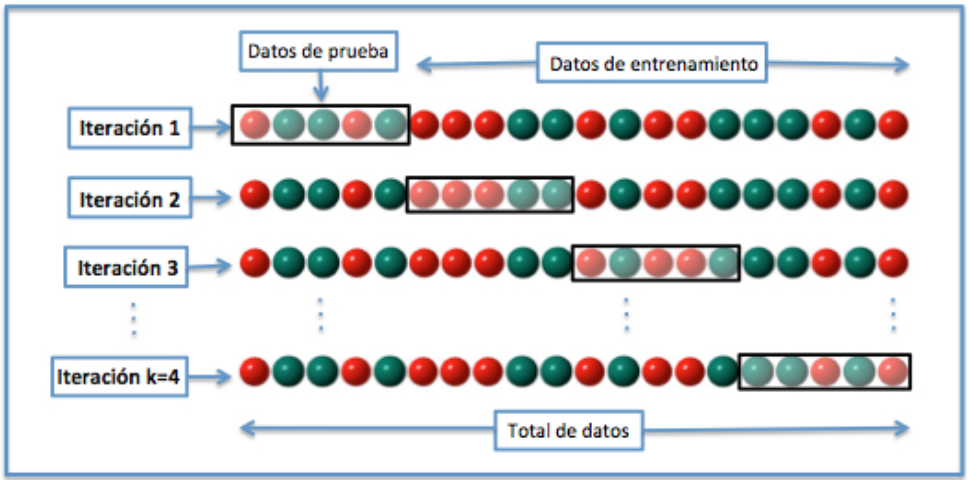


Figura 3.12: Validación cruzada con k = 4 [29]

3.4. Interfaz de usuario

Para la realización de la interfaz gráfica de usuario se ha utilizado la librería ***Kivy*** en Python. Se ha seguido un método híbrido de MVC, donde parte de la vista y del controlador vienen desarrollados en el propio modelo, por comodidad de lectura de código. *Kivy* ofrece programación de la vista en su propio lenguaje, ***KVLang***, que facilita la incorporación de elementos gráficos a la GUI.

También se han desarrollado animaciones en la carga de imágenes para aligerar la interfaz y hacerla más "amigable" al ojo del usuario.

El uso es bastante simple, y puede ser seguido el manual de usuario que se incluye en el apéndice A. Además, se puede seguir los siguientes diagramas, tanto el de flujo (3.13) como el de secuencia (3.14).

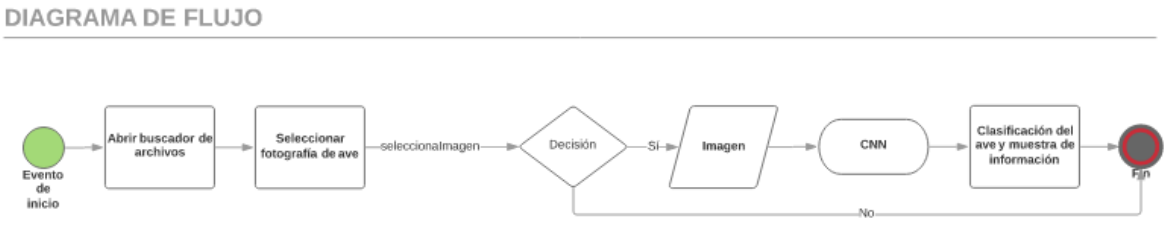


Figura 3.13: Diagrama de flujo de la aplicación

También se ha generado un diagrama de clases para la interfaz de usuario, así como uno que muestra todas los paquetes de la aplicación (figuras 3.15 y 3.16 respectivamente).

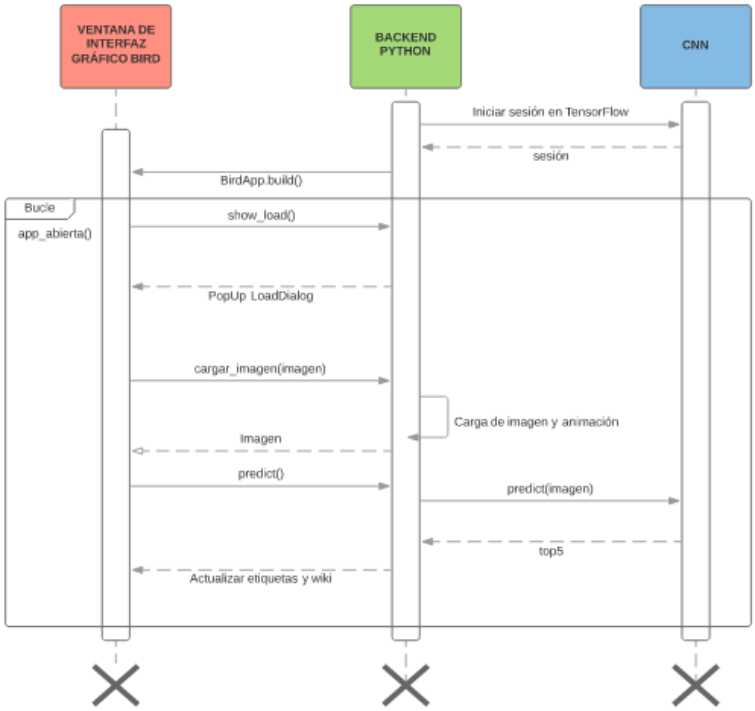


Figura 3.14: Diagrama de secuencia de la aplicación

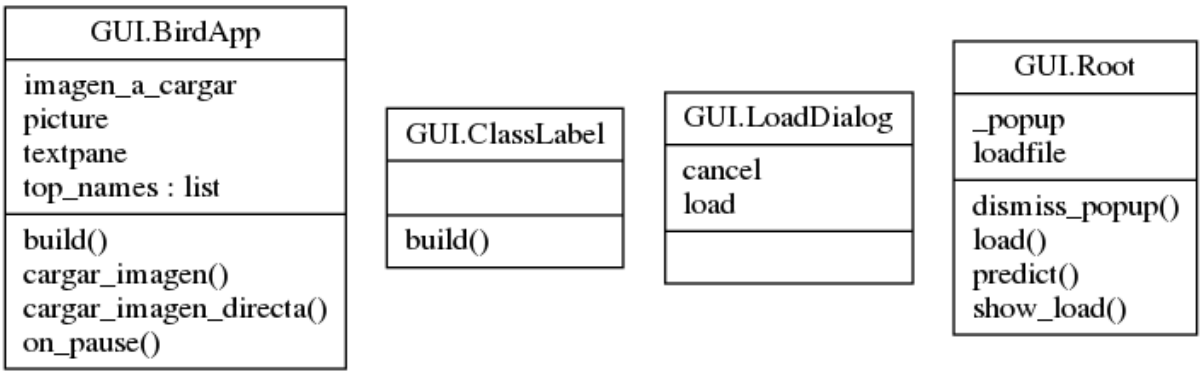


Figura 3.15: Diagrama de clases de la interfaz

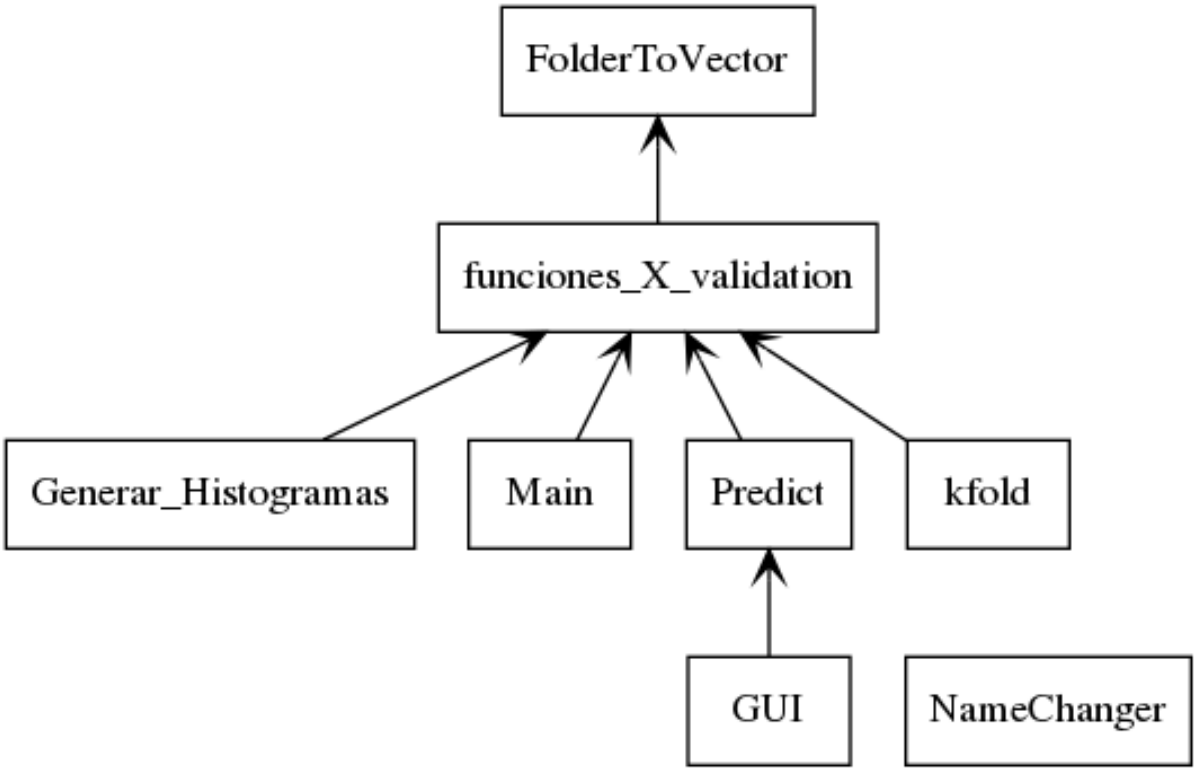


Figura 3.16: Representación de los paquetes de la aplicación

Capítulo 4

Análisis de resultados y discusión

4.1. Resultados del entrenamiento

La medida de precisión que obtenemos es la proporción de aciertos y fallos con respecto al total de imágenes pasadas por la red en la etapa de validación con su respectivo conjunto de validación. Al final de cada entrenamiento se comprueba la precisión final haciendo uso del conjunto de test. Para calcular el error se ha usado la función del **error cuadrático medio (ECM, en inglés *mean squared error* o *MSE*)**. Como su propio nombre indica, se calcula mediante el promedio de los errores al cuadrado, considerando los errores como la diferencia entre el valor real del dato pasado y el valor estimado por la red. La función del error queda representada en la figura 4.1.

$$\mathrm{ECM} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

Figura 4.1: El ECM estimado, siendo $\hat{Y}$ un vector de n predicciones e Y el vector de los valores reales [30]

Los entrenamientos de cada fold se fueron guardando conforme iban entrenando, de manera que se obtuvo una gráfica que expresa la variación de la precisión (*Accuracy*) en entrenamiento (*train*) y validación (*validation*) a lo largo de las épocas que se establecieron (120). El mejor resultado se obtuvo en el fold 4, con una precisión del 91.09 %; el peor, el fold 10, con una precisión del 81.9 %. Los resultados completos se pueden observar en

la tabla 4.2. Para conseguir una precisión más real, se toma como precisión de la red la media aritmética de las precisiones obtenidas en la validación cruzada, dando esto como resultado una precisión media de 86.74 %. Este resultado es bastante positivo para haber tenido un tamaño de conjunto de datos tan reducido, una estratificación de las fotos no óptima (de las aves menos comunes hay menos imágenes, mínimo 30, que de las más comunes como el *Sturnus vulgaris*, con 487 fotografías) y una cantidad de clases diferentes (67) bastante alta para el nº de imágenes del que se disponía. Con ello se quiere decir que los resultados han superado al alza a las expectativas. El entrenamiento finalizó a las 80 horas aproximadamente.

Resultados		
Fold	Precisión	Error (ECM)
1	0.85008	0.00328
2	0.87396	0.00293
3	0.85858	0.00321
4	0.91095	0.00204
5	0.90344	0.00215
6	0.86387	0.00322
7	0.85739	0.00305
8	0.86809	0.00314
9	0.86823	0.00288
10	0.81901	0.00421

Figura 4.2: Tabla comparativa con precisiones y errores de los folds entrenados

En la figura 4.3 se puede observar las diferencias respecto al mejor entrenamiento y el peor, con una diferencia de prácticamente un 10 % de precisión. Se puede observar en el fold nº 10 cómo, sobre la época 40, la gráfica de validación oscila fuertemente sobre el 70 % de precisión. Esto podría ser porque el algoritmo habría caído en un mínimo local. Después de unas 10 épocas oscilando consigue enmendar y retoma la subida de precisión, lo que puede deberse a un cambio en la tasa de aprendizaje (se recuerda que es adaptativa).

El fold nº4 puede observarse que tiende a seguir subiendo. De ser así, podría haber conseguido mayor precisión si hubiera dispuesto de las épocas suficientes. Sin embargo, en el fold nº10 se puede ver que tiende a bajar, lo que, si hubiera continuado, podría haber

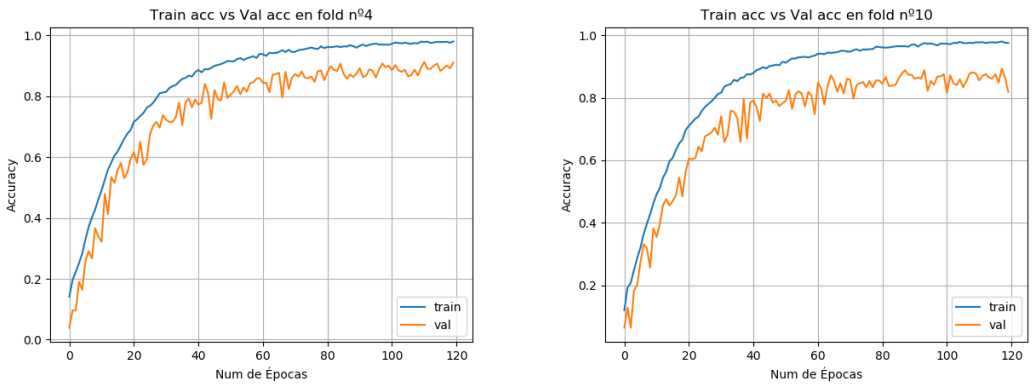


Figura 4.3: Mejor y peor entrenamiento, respectivamente

dado lugar a un sobreajuste de la red.

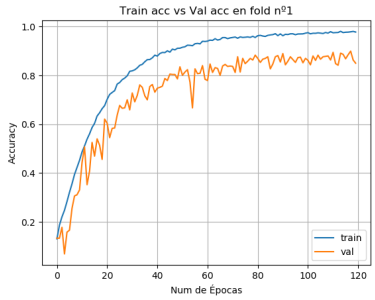
En el resto de entrenamientos se pueden observar las mismas tendencias que podría haber seguido el fold nº4. Sobre todo se puede ver en la precisión como tiende a seguir aumentando, lo que nos podría indicar que con un número mayor de épocas también se hubiera conseguido mejor precisión. Todos los entrenamientos tardaron aproximadamente lo mismo, entre 7.5 y 8 horas.

4.2. Problemas y discusión

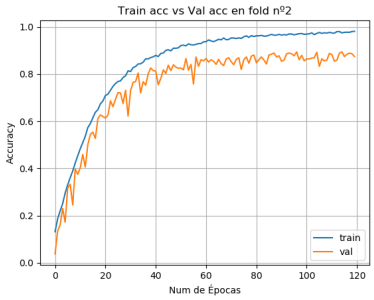
El proyecto ha tenido varios grandes problemas, listados a continuación:

1. Gran número de clases, poca ortogonalidad entre algunas.
2. Número reducido de muestras de entrenamiento.
3. Gran diferencia de luminosidad y colores entre imágenes de la misma clase.
4. Diferencia de número de imágenes entre unas clases y otras.
5. Falta de RAM para ejecutar la validación cruzada sin limpiar ejecuciones anteriores.

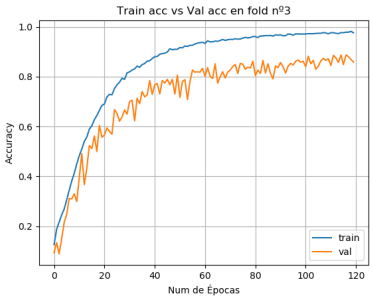
El considerar la falta de ortogonalidad dentro de las clases como un problema surge porque hay aves cuyas características a nivel de color y morfología son similares, como pasa entre las especies *Turdus merula* (sobre todo el macho) y *Sturnus unicolor* (figuras 4.5 y 4.6). Sin embargo, a la hora de realizar la matriz de confusión sobre la red neuronal entrenada (se hizo sobre la que mejor precisión tenía, fold nº4), la red no confunde tanto



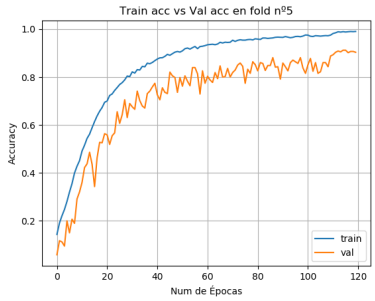
(a) Primer fold



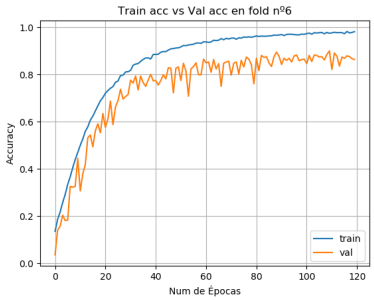
(b) Segundo fold



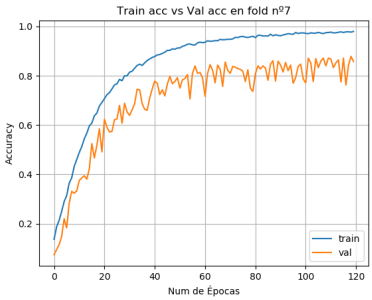
(c) Tercer fold



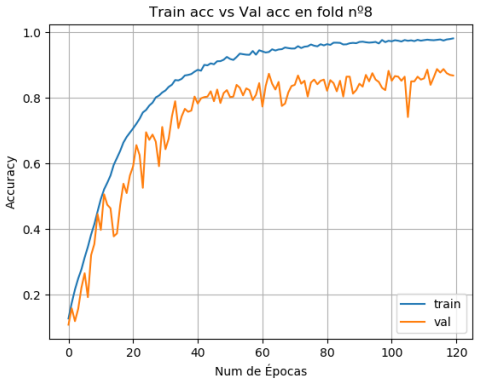
(d) Quinto fold



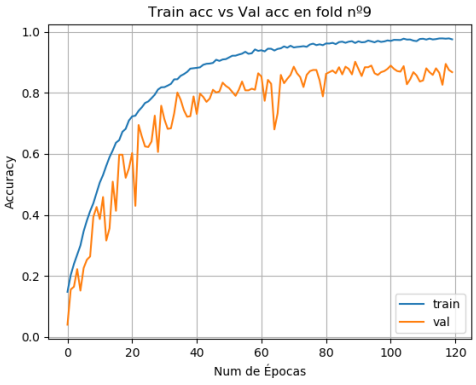
(e) Sexto fold



(f) Séptimo fold



(g) Octavo fold



(h) Noveno fold

Figura 4.4: Gráficas resultado de los folds restantes

Figura 4.5: Mirlo común macho (*Turdus merula*)

Figura 4.6: Estornino negro (*Sturnus unicolor*)

Figura 4.7: Gorrión común macho (*Passer domesticus*)

Figura 4.8: Alcaudón común (*Lanius senator*)

estas dos especies. Las que peor diferenciadas son por ella son *Passer domesticus* macho (figura 4.7) con *Lanius senator* (figura 4.8), como puede observarse en la figura 4.9.

Con ello da lugar a pensar que la visión que tiene la red sobre la información presentada en una imagen no es para nada la misma que tienen los humanos, ya que, por ejemplo, tanto el mirlo como el estornino tienen un tamaño similar, mientras que el alcaudón y el gorrión son completamente distintos en tamaño (ya que el alcaudón es hasta un 50 % más grande que un gorrión). Si en este proyecto se hubieran tomado en cuenta las escalas de los individuos (por ejemplo, tomando siempre las fotos a la misma distancia, podría detectarse el tamaño del ave respecto a otras), posiblemente la red no hubiera tenido tantos problemas clasificando dos especies de distinto tamaño.

Para el problema de la falta de memoria RAM se tuvo que desarrollar un pequeño *script* para ejecutar los entrenamientos de cada fold de manera independiente. Con ello, se consiguió que cada vez que se terminara de ejecutar uno, cerrara la sesión de Tensorflow y Python, limpiando así la memoria de lo que había podido guardar de datos y/o metadatos de las propias sesiones. Las primeras ejecuciones se hicieron en el equipo personal del autor,

que posee 16 GB de memoria RAM, y se ajustó la red para que no se pasase de ese límite. Sin embargo, para realizar la ejecución final de la validación cruzada, teniendo en mente que iba a demorarse bastante (y siendo el equipo un portátil, podría llegar a dañarse por someterlo a altas temperaturas durante un período de tiempo muy prolongado), se tuvo que pasar a ejecutarlo en la máquina descrita como máquina principal al inicio de este capítulo. Como este servidor poseía menor RAM, se tuvo que ajustar el tamaño del paquete de imágenes para cada época, reduciéndolo. Esto, sumado a que la tarjeta gráfica poseía menos frecuencia de reloj, incrementó sustancialmente el tiempo de ejecución con respecto a pruebas anteriores.

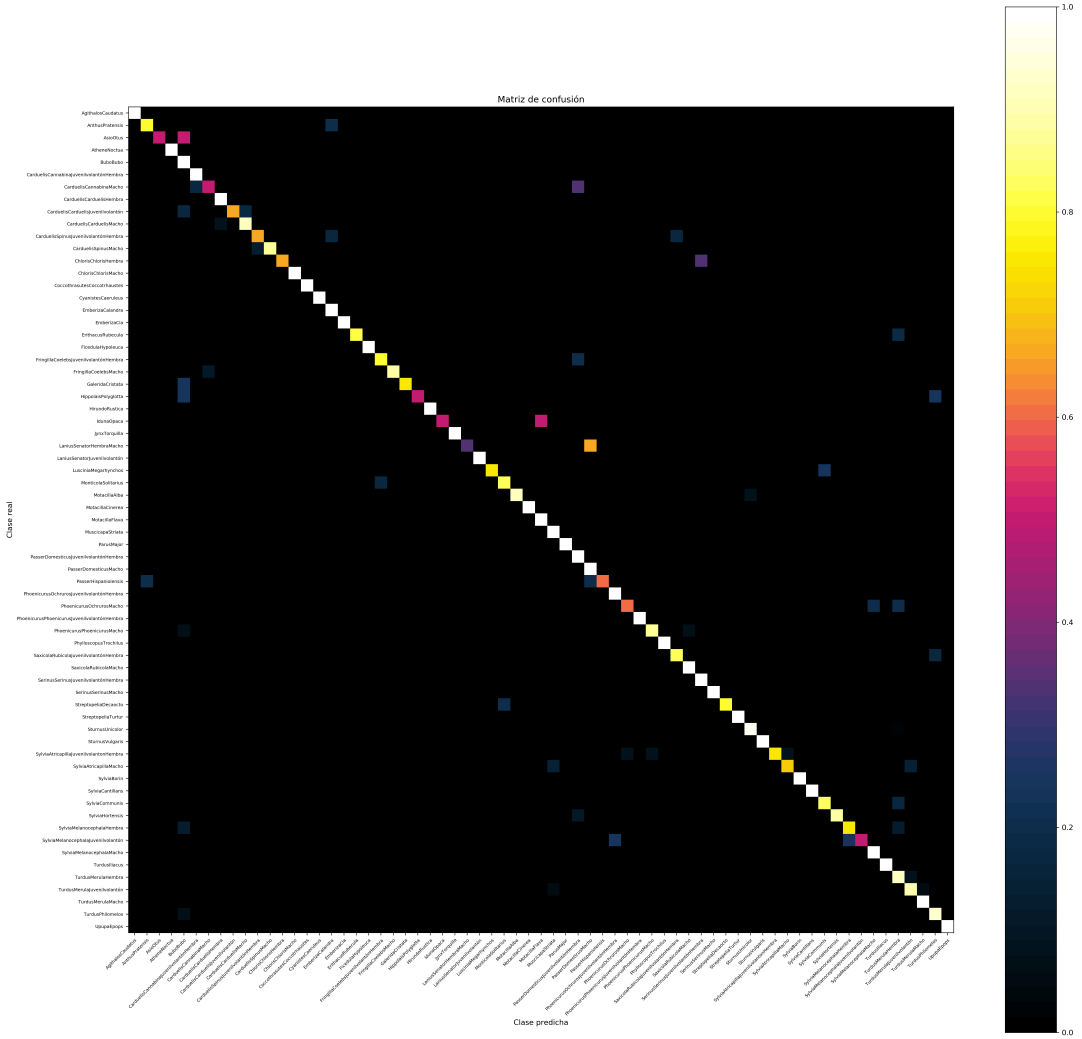


Figura 4.9: Matriz de confusión de la red neuronal (mapa de color sacado de [31])

Capítulo 5

Conclusiones y trabajos futuros

En este proyecto se ha desarrollado un clasificador visual de especies de aves mediante redes neuronales convolucionales, acompañado por un interfaz gráfico de usuario para su uso en la predicción de especies en imágenes que le introduce el actor.

Pese a los múltiples problemas a los que se enfrentó este proyecto, los resultados obtenidos suplen con creces los objetivos establecidos. El cambio de usar una arquitectura lenta y poco precisa como *VGG16* o *VGG19*; a usar *InceptionV3* supuso un cambio sustancial en la precisión de más de un 40 %. El disponer de un conjunto de datos tan limitado no afectó tanto al rendimiento como se hubo pensado en un principio, ya que tener un 91 % de precisión con tal cantidad de clases es un resultado excelente.

Se podría haber obtenido una mejor precisión si se hubiera dispuesto de un número bastante mayor de imágenes y que hubieran estado mejor estratificadas. Sin embargo, como ya se comentó en el párrafo anterior, los resultados obtenidos con menos de 6000 fotos y las más de 60 clases son más que relevantes. Además, el procesamiento de la predicción desde la interfaz de usuario es bastante rápido tras que la sesión en TensorFlow ha quedado iniciada, por lo que se dispone también de una buena eficiencia de predicción.

Ya que se usaron fotografías tomadas directamente de la naturaleza, sin ningún post-tratamiento en ningún tipo de editor, la red se entrenó con imágenes de aves en su estado natural, lo que permite que pueda clasificar fotografías tomadas de la misma forma con facilidad. Esto cumpliría con una de las principales motivaciones de este trabajo, el poder ayudar a aprendices (y no tan aprendices) de ornitología a catalogar aves de las que tengan dudas, o simplemente no conozcan, a las que hayan fotografiado en, por ejemplo, un día de campo.

A partir de lo desarrollado se podría ir mejorando la precisión y el número de clases mediante la recogida de más muestras, lo que le aumentaría el valor sustancialmente. Además, podría desarrollarse una aplicación móvil, la cual se conectara a un servidor con gráfica NVIDIA con el modelo entrenado cargado, de forma que clasificase una imagen tomada con el teléfono y mostrara la información pertinente al usuario, pudiéndose quedar registrada el ave en una base de datos para que actuara a modo de libreta de campo automática.

Bibliografía

[1] Wikipedia, "Anexo sobre las aves de la península ibérica." https://es.wikipedia.org/wiki/Anexo:Aves_de_la_pen%C3%ADnsula_ib%C3%A9rica. Información sobre el conteo de aves en la península ibérica. Accedido última vez marzo, 2019.

[2] SEO, "Carbonero común, Guía de las aves de España." https://www.seo.org/ave/carbonero-comun-2/. Accedido última vez marzo, 2019.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[4] T. Trnovszkỳ, P. Kamencay, R. Orješek, M. Benčo, and P. Sỳkora, "Animal recognition system based on convolutional neural network," *DIGITAL IMAGE PROCESSING AND COMPUTER GRAPHICS*, vol. 15, no. 3, pp. 517–525, 2017.

[5] Google, "Google lens." https://lens.google.com/. Aplicación de clasificación visual desarrollada por Google. Accedido última vez enero, 2019.

[6] TheCornellLab, "Merlin Bird ID." http://merlin.allaboutbirds.org/. Aplicación para la detección de aves. Accedido última vez enero, 2019.

[7] TheCornellLab, "Listados de aves, eBird." https://ebird.org/home. Red social de observaciones de aves. Accedido última vez enero, 2019.

[8] W. He, "Image classification by Keras/CNN for bird species and bird positions detection (By FRCNN)." https://github.com/offthewallace/Summer-2017 and http://www.mathcs.richmond.edu/~lbarnett/research/summer2017/index.php. Aplicación modelo para este trabajo. Accedido última vez diciembre, 2018.

[9] SEO, "SEO, clasificador de aves." https://www.seo.org/2016/12/27/partir-ahora-no-podras-decir-no-conoces-pajaro/. Clasificación de aves por selección de características. Accedido última vez enero, 2019.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] A. Cartas, "Trabajo propio, CC BY-SA 4.0." https://commons.wikimedia.org/w/index.php?curid=41534843. Diagrama de un perceptrón con cinco señales de entrada. Accedido última vez mayo, 2019.

[12] ml4a, "Machine Learning for artists, Neural networks." https://ml4a.github.io/ml4a/es/neural_networks/. Funciones de activación. Accedido última vez mayo, 2019.

[13] Wikipedia, "Vanishing gradient problem." https://en.wikipedia.org/wiki/Vanishing_gradient_problem. Problemas de usar el gradiente descendente con la función sigmoide. Accedido última vez mayo, 2019.

[14] C. S. Wiki, "Max-Pooling." https://computersciencewiki.org/index.php/Max-pooling_/_Pooling. Ejemplo max pooling. Accedido última vez mayo, 2019.

[15] SuperDataScience, "Convolutional Neural Networks (CNN): Step 3 - Flattening." https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening. Accedido última vez mayo, 2019.

[16] D. Mwiti, "Convolutional neural networks: An intro tutorial." https://heartbeat.fritz.ai/a-beginners-guide-to-convolutional-neural-networks-cnn-cf26c5ee17ed. Accedido última vez mayo, 2019.

[17] R. B. Zadeh and B. Ramsundar, "Fully Connected Deep Networks." https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html. Accedido última vez mayo, 2019.

[18] Wikipedia, "Softmax function." https://en.wikipedia.org/wiki/Softmax_function. Accedido última vez junio, 2019.

[19] R. Prabhu, "Understanding of Convolutional Neural Network (CNN) Deep Learning." https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148. Accedido última vez mayo, 2019.

[20] E. Culurciello, "A Brief History Of Neural Network Architectures." https://www.topbots.com/a-brief-history-of-neural-network-architectures/. Accedido última vez mayo, 2019.

[21] P. Capper, "Hawkes Bay, Nueva Zelanda." https://commons.wikimedia.org/wiki/File:Unequalized_Hawkes_Bay_NZ.jpg. Accedido última vez enero, 2019.

[22] H. Everding, "CIELAB color space front view." https://commons.wikimedia.org/wiki/File:CIELAB_color_space_front_view.png. Accedido última vez enero, 2019.

[23] Wikipedia, "Adaptive histogram equalization." https://en.wikipedia.org/wiki/Adaptive_histogram_equalization. Accedido última vez enero, 2019.

[24] U. de Alicante, "Modelo vista controlador (MVC)." https://si.ua.es/es/documentacion/asp-net-mvc-3/1-dia/modelo-vista-controlador-mvc.html. Accedido última vez mayo, 2019.

[25] Wikipedia, "Modelo vista controlador (MVC)." https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador. Accedido última vez mayo, 2019.

[26] A. Milton-Barker, "Inception V3 Deep Convolutional Architecture For Classifying Acute Myeloid/Lymphoblastic Leukemia." https://software.intel.com/en-us/articles/inception-v3-deep-convolutional-architecture-for-classifying-acute-myeloidlymphob Accedido última vez mayo, 2019.

[27] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[28] V. Bushaev, "Understanding RMSprop, faster neural network learning." https://towardsdatascience.com/

understanding-rmsprop-faster-neural-network-learning-62e116fcf29a. Accedido última vez mayo, 2019.

[29] J. Domenech, "Validación cruzada de k iteraciones." https://commons.wikimedia.org/wiki/File:K-fold_cross_validation.jpg. Accedido última vez mayo, 2019.

[30] Wikipedia, "Error cuadrático medio." https://es.wikipedia.org/wiki/Error_cuadr%C3%A1tico_medio. Accedido última vez mayo, 2019.

[31] M. Geissbuehler and T. Lasser, "How to display data by color schemes compatible with red-green color perception deficiencies," *Optics express*, vol. 21, no. 8, pp. 9862–9874, 2013.

Apéndice A

Manual de usuario

La instalación de dependencias se hace mediante la importación de un entorno de anaconda3. Para importar el entorno, junto al archivo .yml, hay que ejecutar la siguiente sentencia en terminal:

```
conda env create -f environment.yml
```

Tras ello, hay que activar el entorno para poder ejecutar la aplicación:

```
source activate tensorflow
```

Por último, instalar la librería que extrae información de wikipedia:

```
pip install wikipedia-api
sudo apt-get install xclip xsel
```

Tras ello, la aplicación ya podrá ser lanzada.

A.1. Entrenamiento

Para probar el entrenamiento, deberá ejecutar el siguiente comando:

```
sh ejecucion.sh
```

Hará falta un mínimo de 8 GB de memoria RAM para funcionar. Los pesos se guardarán en cada una de las carpetas pertinentes a cada fold.

A.2. Uso de la aplicación

Para lanzar la aplicación, hay que ejecutar:

```
python GUI.py
```

tal y como se muestra en la figura A.1, esa debería ser la salida por terminal; y la aplicación se abriría directamente.

```
(tensorflow) antonio@antonio-GE73-Raider-RGB-8RF:~/Documentos/Universidad/Cuarto/TFG/Definitivo$ python GUI.py

Using TensorFlow backend.
2019-06-09 10:34:20.080119: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instruction
s that this TensorFlow binary was not compiled to use: SSE4.1 SSE4.2 AVX AVX2 FMA
2019-06-09 10:34:20.386521: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:897] successful NUMA node r
ead from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2019-06-09 10:34:20.387316: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1405] Found device 0 with prope
rties:
name: GeForce GTX 1070 major: 6 minor: 1 memoryClockRate(GHz): 1.695
pciBusID: 0000:01:00.0
totalMemory: 7.93GiB freeMemory: 7.65GiB
2019-06-09 10:34:20.387350: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1484] Adding visible gpu device
s: 0
2019-06-09 10:34:25.803889: I tensorflow/core/common_runtime/gpu/gpu_device.cc:965] Device interconnect Stream
Executor with strength 1 edge matrix:
2019-06-09 10:34:25.803955: I tensorflow/core/common_runtime/gpu/gpu_device.cc:971]      0
2019-06-09 10:34:25.803976: I tensorflow/core/common_runtime/gpu/gpu_device.cc:984] 0:   N
2019-06-09 10:34:25.813330: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1097] Created TensorFlow device
 (/job:localhost/replica:0/task:0/device:GPU:0 with 4059 MB memory) -> physical GPU (device: 0, name: GeForce
GTX 1070, pci bus id: 0000:01:00.0, compute capability: 6.1)
[INFO   ] [Logger      ] Record log in /home/antonio/.kivy/logs/kivy_19-06-09_0.txt
[INFO   ] [Kivy        ] v1.10.1
[INFO   ] [Python      ] v3.6.7 |Anaconda, Inc.| (default, Oct 23 2018, 19:16:44)
[GCC 7.3.0]
[INFO   ] [Factory     ] 194 symbols loaded
[INFO   ] [Image       ] Providers: img_tex, img_dds, img_sdl2, img_pil, img_gif (img_ffpyplayer ignored)
[INFO   ] [Window      ] Provider: sdl2(['window_egl_rpi'] ignored)
[INFO   ] [GL          ] Using the "OpenGL" graphics system
[INFO   ] [GL          ] Backend used <gl>
[INFO   ] [GL          ] OpenGL version <b'4.6.0 NVIDIA 410.48'>
[INFO   ] [GL          ] OpenGL vendor <b'NVIDIA Corporation'>
[INFO   ] [GL          ] OpenGL renderer <b'GeForce GTX 1070/PCIe/SSE2'>
[INFO   ] [GL          ] OpenGL parsed version: 4, 6
[INFO   ] [GL          ] Shading version <b'4.60 NVIDIA'>
[INFO   ] [GL          ] Texture max size <32768>
[INFO   ] [GL          ] Texture max units <32>
[INFO   ] [Window      ] auto add sdl2 input provider
[INFO   ] [Window      ] virtual keyboard not allowed, single mode, not docked
[INFO   ] [Text        ] Provider: sdl2
[INFO   ] [GL          ] NPOT texture support is available
xclip version 0.12
Copyright (C) 2001-2008 Kim Saunders et al.
Distributed under the terms of the GNU GPL
[INFO   ] [Clipboard   ] Provider: xclip
[INFO   ] [CutBuffer   ] cut buffer support enabled
[INFO   ] [ProbeSysfs  ] device match: /dev/input/event5
[INFO   ] [MTD         ] Read event from </dev/input/event5>
[INFO   ] [Base        ] Start application main loop
[WARNING] [MTD         ] Unable to open device "/dev/input/event5". Please ensure you have the appropriate per
missions.
```

Figura A.1: Salida por terminal de la ejecución

Lo siguiente, para abrir una imagen, habría que pulsar el botón de añadir, situado en la esquina inferior izquierda, tal y como se puede observar en la figura A.2.

Tras esto, se abriría una pantalla de selección de archivo, como puede observarse en la figura A.3.

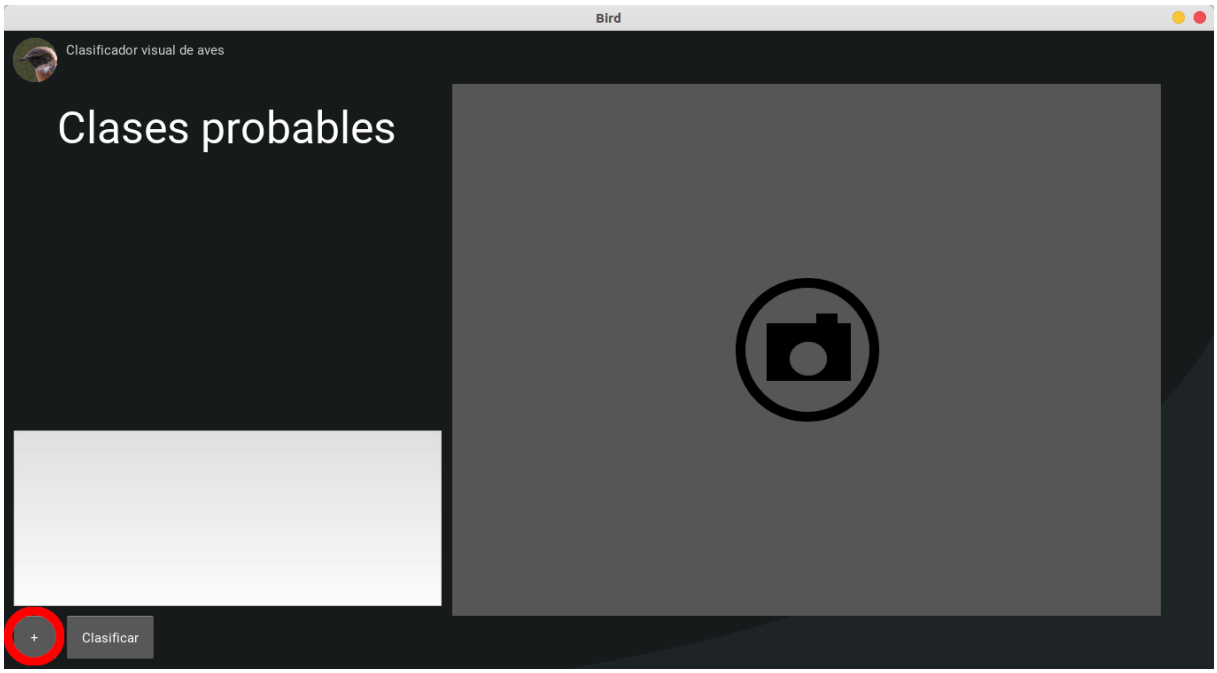


Figura A.2: Aplicación recién abierta

Hay que navegar por las carpetas hasta encontrar la imagen del ave a clasificar, siguiendo como guía al encontrarla la figura A.4

Una vez cargada la imagen, tras una pequeña animación ya habrá terminado de prepararla. Tras esto, hacer click en el botón de Clasificar tal y como se puede observar en la figura A.5.

Tras un breve lapso de tiempo, la imagen es clasificada y se muestra en el cuadro de texto un resumen de la especie con mayor probabilidad desde la página de Wikipedia de la misma (figura A.6).

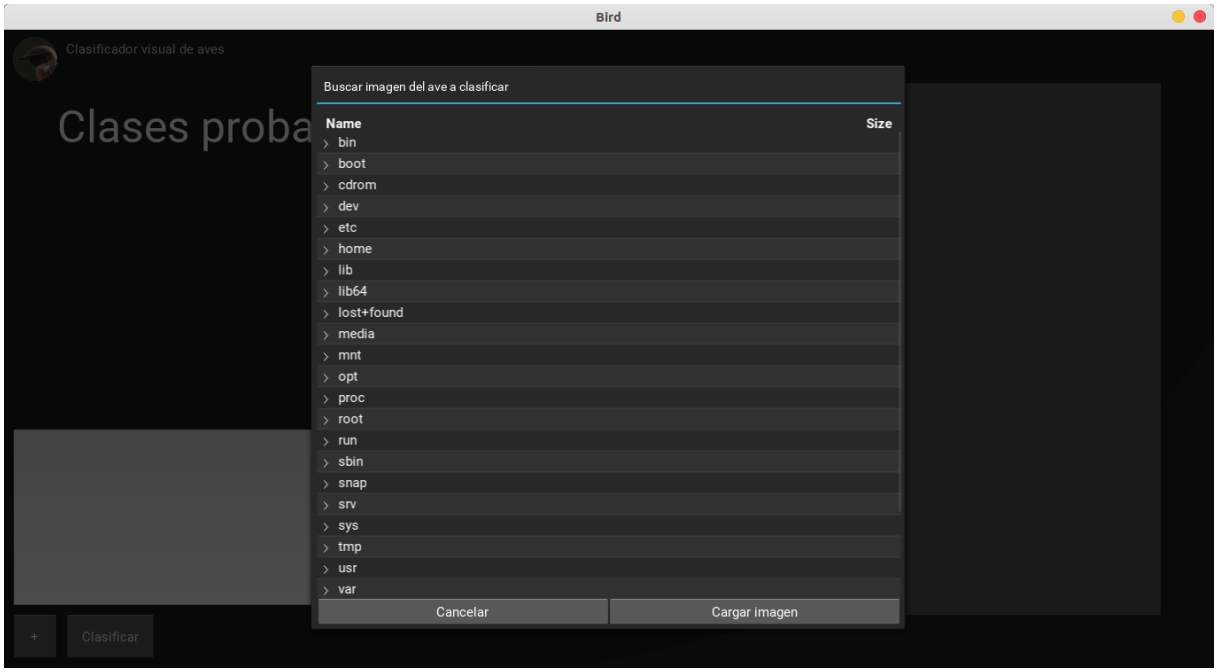


Figura A.3: Pantalla de selección de imagen

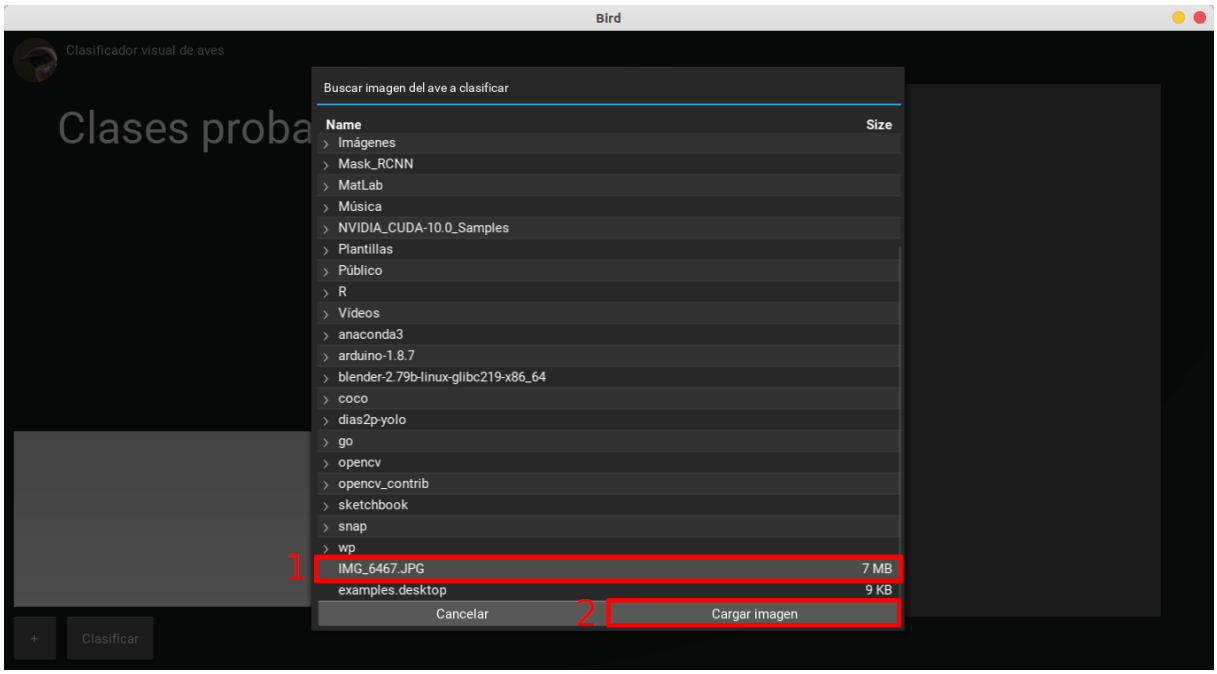


Figura A.4: Selección final de imagen

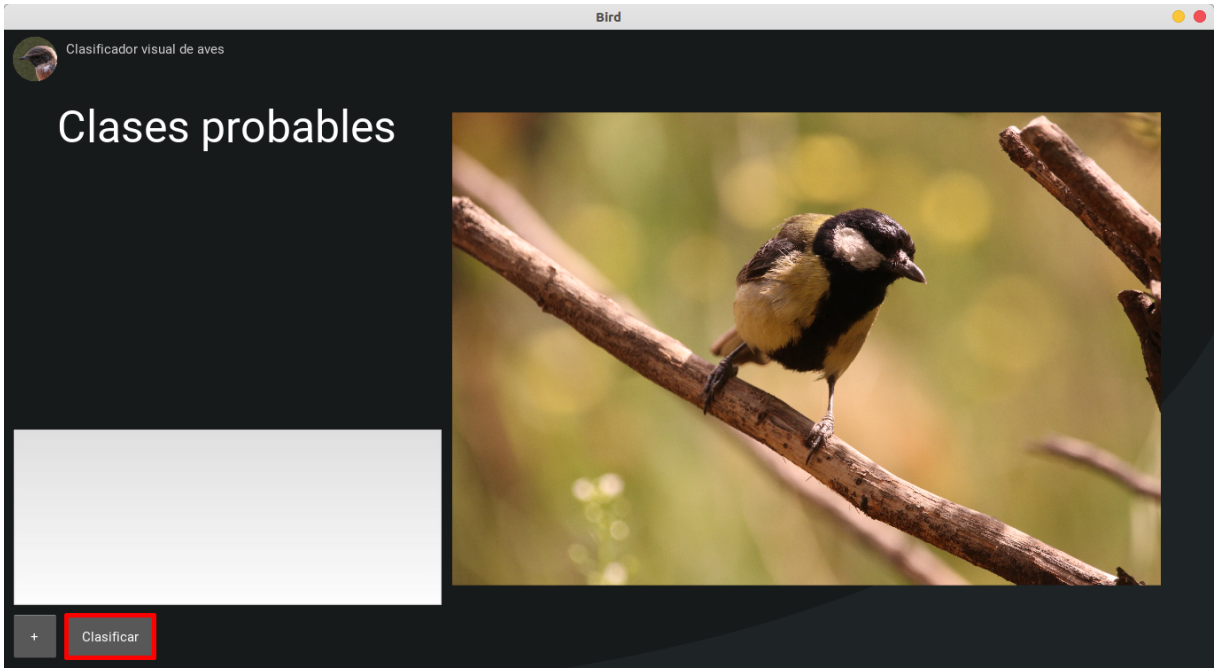


Figura A.5: Clasificación de la imagen

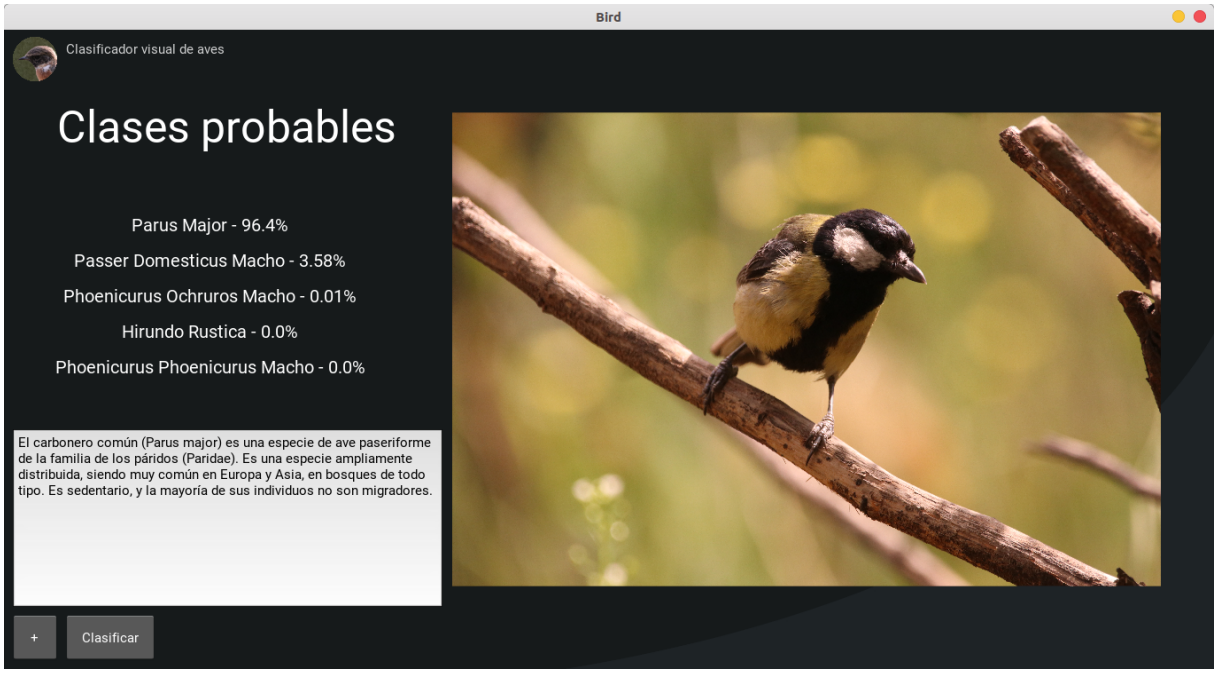


Figura A.6: Finalización de la clasificación de la imagen